**MASTER EN SCIENCES INFORMATIQUES**

# Etude, évaluation et utilisation du paradigme de programmation orienté aspect (AOP)

Pirart, Gaël

*Award date:*
2005

Facultés Universitaires Notre-Dame de la Paix, Namur
Institut Informatique
Année académique 2004-2005

# Étude, évaluation et utilisation du paradigme de programmation orienté aspect (AOP).

**Auteur du Mémoire: Gaël Pirart**

Mémoire présenté en vue de l'obtention du grade de Licencié en Informatique.

## Résumé

Le paradigme de programmation orienté objet est devenu fort populaire car ses concepts ont permis d'améliorer la compréhension des projets informatiques et à mieux en gérer la complexité grâce, entre autres, aux facilités de réutilisation des composants. Cependant, dans la pratique, de nombreuses difficultés se dressent encore lorsque des modifications doivent être apportées à une application de taille importante.

Le paradigme de programmation orienté aspect (AOP) recule les barrières de cette complexité en venant se greffer au paradigme objet. Le code source de la logique métier obtenu est exempt de toute référence aux aspects qui sont le résultat de l'implémentation de préoccupations telles que la gestion de la persistance, de l'authentification, des transactions, ...
Le gain en modularité se traduit par la possibilité de remplacer une partie localisée du code ou même un composant entier par un autre, sans affecter les parties indépendantes de ce changement.

Le mémoire aborde AOP par ses concepts et ses outils, puis concrétise le sujet par l'analyse et le design d'un cas réel. Cette approche offre une vue globale du paradigme, laissant pressentir dans quelles mesures cette technologie influera sur le monde de l'ingénierie logicielle de demain.



**Mots-clés:** Programmation orientée aspect, tissage, outils, analyse, AOSD, serveurs d'applications.



## Abstract

Object oriented programming paradigm has gained popularity thanks to its main concepts. It provides a better comprehension to IT projects and facilitates the complexity management with the ability to reuse components. However, many issues still occur when changes must be brought to a large scale application.

Aspect oriented paradigm (AOP) is a new step in the domain of complexity management. It groups crosscutting concerns source code in aspects, so the application's logic source code does not have any reference to these concerns anymore (like persistency, authentication, transactions ...).
It results in a more modular application where the developer is able to replace a local part of code or even an entire component, without any impact on the independent part of this change.

This work explores AOP first by introducing its concepts and tools. The second part is a concrete case of project, through its analysis and design. This kind of approach gives a global view of the paradigm, and offers the ability to sense how it will change the software engineering world of tomorrow.



**Keywords:** aspect oriented programming, weaving, tools, analysis, AOSD, applications server.

# Avant-propos

Il est difficile de se rendre compte réellement de l'aventure que ce type de travail représente avant de l'avoir vécu soi-même.

La recherche de documentation est ponctuée par des remises en questions sur les acquis, entraînant souvent plus de questions que de réponses. Puis, à mesure que le temps passe et que le travail avance, la masse d'informations récoltées s'ordonne et se complète. La pratique permet de dissiper les derniers doutes. Au final, tout n'a pas été exploré, mais il faut bien s'arrêter un jour.

Tout cela n'aurait pu aboutir sans le soutien de certaines personnes que je tiens à remercier:
- Patrick Heymans, le promoteur de ce mémoire, pour le temps investi, ses critiques, ses conseils et sa sympathie.
- Ma femme Aude et mon fils Arthur pour la joie et la motivation qu'ils m'apportent tous les jours.
- Les responsables de la sociéte Micro Research pour m'avoir soutenu dans ces études.
- L'ensemble des acteurs de la faculté informatique à horaire décalé sans qui je n'aurais jamais pu vivre cette expérience.
- Et enfin à mon père qui m'a mis derrière le clavier d'un Commdore64 alors que je n'avais pas 7ans.

# Table des matières

# Glossaire

**Annotations** : Les annotations sont des tags (mnémoniques) pouvant être définis pour enrichir le langage Java. Ce système d'annotations est disponible par défaut à partir de la version 1.5 de Java.

**Application Programming Interface (API)** : Ensemble des interfaces décrivant les méthodes accessibles au programmeur pour utiliser une librairie par exemple.

**Aspect** : Résultat du groupement du code relatif à une préoccupation du développeur qui serait normalement dispersé et redondant au travers des classes d'une l'application. Les interactions entre l'application et l'aspect sont définies dans ce dernier de telle sorte qu'aucune référence à l'aspect ne soit visible dans le code source de l'application.

**Aspect Oriented Programming (AOP)** : Voir à 'Programmation Orientée Aspect'.

**Benchmarking** : Voir la définition au Chapitre 2, Section 1, en page 25.

**Broker** : Composant dont le rôle est de permettre l'échange de messages entre des composants utilisant des protocoles de communication différents.

**Bytecode** : Code obtenu à l'aide du compilateur Java à partir du code source. Le bytecode peut ensuite être interprété par la JVM lors de son exécution.

**Classloader** : Classe de l'API Java ayant pour fonction de chercher et de charger d'autres classes.

**Commercial off-the-shelf** : Produit utilisé à des fins commerciales, dont la caractéristique principale est de s'adapter au besoin du client par quelques changement de configuration.

**Container** : Nom donné au composant qui a pour charge d'instancier des classes, de les mettre à disposition et de les détruire. Par exemple, un container d'EJB's.

**Design Pattern** : Voir à 'Patron de Conception'.

**Enterprise Java Bean (EJB)** : Classe Java disposant de services spécifiques (transaction, sécurité, persistance, clustering, ...) définies par les normes J2EE. Le cycle de vie d'un EJB est régi par son container.

**Fonctionnalité transversale** : Fonctionnalité secondaire qui est utilisée par la fonctionnalité principale relative à la logique métier. Par exemple, un composant de gestion comptable (fonctionnalité principale) a besoin comme de sauvegarder ses données (fonctionnalité secondaire).

**Framework d'application** : Voir 'serveur d'applications'.

**Gestion des traces** ('logging' en anglais) : "Positionnement de messages à différents endroits du code destinés à être affichés à l'écran ou stockés dans une structure de données" [Pawlak et al. 04].
Voir également la définition au Chapitre 2, Section 1, en page 25.

**Injection de code** : Technique de développement qui consiste à fournir à une instance les autres instances dont elle a besoin plutôt qu'elle ne les instancie elle-même. Cela permet d'améliorer la modularité du code de la classe car elle ne travaille qu'avec des interfaces.

**Integrated Development Environment (IDE)** : C'est un outil développement qui offre de nombreuses facilités tels que l'éditeur, le débogueur, la compilation automatique du code, le contrôle de versions, des outils d'exploration des structures de données et du code, etc.

**Inversion Of Control (IoC)** : Voir à 'Injection de code'.

**Javadoc** : Commentaires écrits dans le code qui sont formalisés de manière à générer automatiquement la documentation des sources en html.

**Java 2 Platform, Entreprise Edition (J2EE)**: Standard de la société Sun qui définit les spécifications normalisées d'un serveur d'applications multicouches et distribuées en Java.

**Java Virtual Machine (JVM)** : Le code source en Java devient du bytecode après avoir été compilé. La machine virtuelle de Java est le programme capable d'interpréter le bytecode.

**Logging** : Voir à 'Gestion des traces'.

**Machine virtuelle** : Voir à 'Java Virtual Machine'.

**Model – View – Controller (MVC)** : Architecture logicielle pour la conception de systèmes informatiques. Elle vise principalement à séparer la logique métier (Model) du composant de visualisation (View) par l'intermédiaire d'un composant de contrôle (Controller).

**Monitoring** : la définition au Chapitre 2, Section 1, en page 25.

**Operations Support System for Java (OSS/J)** : Norme de standardisation des interfaces permettant de faciliter l'interconnexion entre des composants Java dans le monde des télécoms.

**Package** : Nom donné au répertoire dans lequel se trouve une classe Java.

**Patron de conception** : techniques de programmation standardisées permettant de résoudre des problèmes récurrents dans le monde de l'ingénierie logicielle.

**Préoccupation** (concern en anglais) : Une préoccupation est un domaine utilisé comme critère de décomposition pour un système [Czarnecki 97]. De cette préoccupation résulte le besoin d'implémenter un service, une fonctionnalité. Par exemple, la persistance (sauvegarde) des objets dans une base de données relationnelle est un exemple de préoccupation.

**Préoccupation transversale** : voir fonctionnalité transversale.

**Programmation Orientée Aspect (POA)** : voir définition au Chapitre 1, Section 2, en page 6.

**Proxy** : Composant servant de façade à un autre objet. Il sert le plus souvent à cacher le fait que l'objet ciblé réellement soit distant.

**Proxy dynamique** : Proxy qui a la particularité d'implémenter des interfaces spécifiées à l'exécution, au moment de sa création.

**Serveur d'applications** : Un serveur d'applications est un système sur lequel sont installées des applications. Elles peuvent ainsi profiter de services fournis par le serveur. Ces services sont accessibles généralement via un API.

**Singleton** : Une classe ayant cette propriété n'a qu'une seule instance en mémoire à la fois.

# Acronymes

**API** : Application Programming Interface.
**AOP** : Aspect Oriented Programming.
**BNM** : Broadband Network Management.
**COTS** : Commercial off-the-shelf
**EJB** : Enterprise Java Bean
**GUI** : Graphical User Interface
**IDE** : Integrated Development Environment.
**IoC** : Inversion of Control
**J2EE** : Java 2 Platform, Entreprise Edition.
**JVM** : Java Virtual Machine
**MVC** : Model – View – Controller.
**OSS/J** : Operations Support System for Java
**POA** : (AOP en anglais) Programmation Orientée Aspect.
**POO** : (OOP en anglais) Programmation Orientée Objet.

# Introduction

## Buts du mémoire

Le paradigme de programmation orientée aspect est récent: il est seulement apparu dans le monde de l'ingénierie logicielle à la fin des années nonante. Il se base sur le concept d'aspect dans le but de répondre aux besoins actuels de modularité dans la conception et l'évolution d'applications orientées objets.

Ce travail se veut d'être une analyse critique de la technologie de l'aspect. Le sujet est abordé sous de nombreux angles dans l'intention de réunir suffisamment d'éléments significatifs tant sur le plan théorique que pratique, sur base d'un projet réel. Dans ces conditions, il devient possible de conclure quels sont les apports de l'aspect face aux autres solutions apportant des moyens de modularité accrus, et quelles sont les contraintes qui en découlent.

Ce condensé d'informations se doit surtout de fournir au lecteur une initiation au monde de l'aspect, de lui donner une vision objective de l'utilité de cette technologie et cela, dans le contexte actuel et futur.

## Le mémoire au service du projet

Micro Research S.A. est une société de services informatiques spécialisée dans le domaine des télécommunications. Le projet BNM qu'elle réalise a pour but de créer une application capable de gérer le parc des éléments de réseaux IP de sociétés de télécom.

BNM comprend plusieurs volets. En voici les principaux:

- la détection des éléments du réseau,
- la persistance de la topologie du réseau,
- la configuration des éléments de réseaux[1],
- la possibilité de configurer des services complexes sous forme de workflows.

L'application BNM est construite sur base du framework d'applications JBoss. JBoss est une implémentation des spécifications J2EE définissant les spécifications normalisées d'un serveur d'applications distribuées Java.

Afin de vérifier le bon fonctionnement de l'application, d'en améliorer les performances et d'identifier les besoins au niveau système, un des buts du projet est de mettre en place les fonctionnalités de logging, de monitoring et de benchmarking. Le logging permet de suivre l'acheminement des requêtes utilisateurs, le monitoring de mesurer la charge en ressources système prise par les composants de l'application, et le benchmarking de comparer les performances enregistrées au monitoring de différentes configurations système (software & hardware).

La programmation orientée aspect est un paradigme émergeant qui a la réputation de répondre parfaitement aux besoins de ce type de fonctionnalité (logging, monitoring & benchmarking). Afin de connaître l'apport de cette nouvelle technologie dans le monde informatique, les responsables du projet BNM ont donné leur aval pour que ce mémoire porte sur cette partie du projet. Le but de l'entreprise

[1] La tâche de configuration des éléments de réseau est appelée 'provisioning' dans le monde des télécoms.

est d'évaluer et d'identifier quels sont les outils AOP existants utilisables dans le cadre du projet. Si les outils remplissent les conditions du cahier des charges, il s'agit d'utiliser le plus adapté pour implémenter la solution. Dans le cas contraire, une implémentation alternative, plus classique est à prévoir.
En plus de l'évaluation et de l'implémentation, les responsables ont demandé d'effectuer l'ensemble de la tâche d'analyse et de design de ces fonctionnalités. Si le choix de l'aspect est adopté, il est possible de mesurer les incidences du paradigme aspect sur l'entièreté d'un processus de conception d'un projet. Dans le cas contraire, le mémoire portera sur la mise en place d'une autre solution de modularité qui respecte les contraintes du cahier des charges.

L'ensemble de l'évaluation des outils AOP, de l'analyse et de la conception du projet BNM pour la partie logging, monitoring et benchmarking vous est donc présenté dans ce travail.

## Contenu du mémoire

Ce document est divisé en 2 chapitres.

Le premier présente le paradigme de programmation orienté aspect au travers de ses concepts et de ses outils. Le deuxième chapitre est d'une approche plus pragmatique. Il présente l'analyse et la conception du système développé pour l'application BNM.

La conclusion fournit la critique de l'auteur sur l'expérience que lui a apportée cette étude.

*Quelques remarques préliminaires:*

- Il est supposé que le lecteur dispose déjà d'une bonne compréhension du paradigme de programmation orienté objet et plus particulièrement de Java. En plus d'être gratuit, l'environnement Java/J2EE est un monde particulièrement actif dans la prolifération d'outils AOP.
- Certains termes techniques sont consciemment maintenus en anglais, principalement ceux dont la traduction française est peu usitée dans le microcosme de l'informatique.

# Chapitre 1. Le paradigme orienté aspect

## Section 1. Constat sur le paradigme orienté objet

La première étape de l'élaboration d'un logiciel peut être vue comme une activité de réduction de problème, où l'on décompose des problèmes en sous-problèmes plus faciles à résoudre [Czarnecki 97]. Lorsqu'ils sont isolés et identifiés, leurs solutions sont déduites et modélisées. Puis, finalement, chaque solution envisagée doit être traduite dans le langage de programmation désiré.

Contrairement à d'autres paradigmes qui imposent une traduction dans un langage totalement indépendant du problème posé, le paradigme de programmation orientée objet (POO) "conserve la structure du domaine". Le POO permet de représenter les concepts du domaine en classes, reléguant l'architecture imposée par la machine à un niveau secondaire [Eckel 00]. Cette abstraction de type objet permet de respecter la typologie du problème jusqu'à son implémentation, ce qui est bénéfique pour la compréhension du code développé.

Les classes obtenues décrivent chacune l'ensemble des données et des traitements propres à un type déterminé.

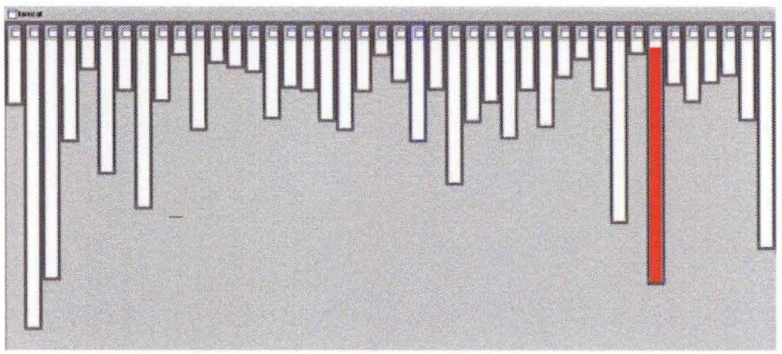


**Figure 1. Classe chargée de l'analyse syntaxique XML dans org.apache.tomcat [Kersten 02].**

Voici un exemple. La figure1 illustre l'implémentation des classes de l'application Tomcat, où chaque bâtonnet représente une classe. La classe colorée regroupe l'ensemble des données et méthodes lié à l'analyse syntaxique en XML.

L'encapsulation, l'héritage, la surcharge et le polymorphisme sont des notions qui régissent les relations entre classes. Elles contribuent à mieux structurer le code source. Les voici classées selon les avantages qu'elles peuvent apporter au programmeur dans [Czarnecki 97] :

- **L'apport d'une meilleure compréhension et d'une meilleure gestion de la complexité.**

En plus de la notion d'objet qui aide à construire des modèles plus compréhensifs, l'encapsulation améliore les possibilités d'abstraction, et l'héritage organise les classes de manière hiérarchique.

- **L'extensibilité, l'adaptabilité et la réutilisation.**

Les mécanismes d'héritage, de surcharge et le polymorphisme permettent des implémentations alternatives de composants.

- **La maintenance**

L'encapsulation aide à réduire le couplage tandis que le polymorphisme permet d'insérer de nouveaux types d'objets avec un impact limité sur le code.

Malgré ses attraits, la POO ne paraît pas être une solution totalement satisfaisante pour obtenir une décomposition optimale et totalement modulaire, notamment aux niveaux des fonctionnalités transversales ou de la dispersion de code [Pawlak et al. 04].
Lorsque l'on décrit une fonctionnalité au travers d'une classe, il est courant que celle-ci interagisse avec d'autres fonctionnalités plus ou moins indépendantes. On les appelle des **fonctionnalités transversales**.
Illustrons ce constat à l'aide d'un exemple: la classe Compte. Sa fonctionnalité principale est la gestion d'un compte en banque. Pour partir d'un exemple simple, considérons qu'un compte n'a pour donnée que son solde et que l'on ne puisse y effectuer que des retraits.
Le code source (en Java) résultant est présenté en Figure 2.

```
package gestion;

/**
 * Gestion de Compte
 */
public class Compte {

    long solde = 0;

    Compte(long soldeInitial) {
        solde = soldeInitial;
    }

    void retrait(long montant) throws Exception {

        if ( (solde-montant) < -1000 ) {
          /* Cas de refus */
            throw new Exception("Solde insuffisant!");
        } else {
            /* Cas de réussite */
            solde= solde-montant;
        }
    }
}
```


**Figure 2. Classe Compte et son diagramme de classes.**

Imaginons que la banque ait besoin de garder un historique des opérations effectuées sur ce compte. On utilise dans ce cas un outil de gestion de traces dont le rôle est d'enregistrer et de filtrer les messages retraçant les évènements survenant lors des opérations.
Pour notre exemple, admettons que notre programmeur choisisse log4j [log4j 05] comme outil de gestion de traces. Il utilise l'API de log4j pour signaler chaque opération de retrait, son échec ou sa réussite.
La Figure 3 présente la nouvelle version de la classe Compte.

La classe Compte, fonctionnalité de base, utilise la gestion de traces afin de gérer l'historique des opérations bancaires. On est donc en présence d'une fonctionnalité transversale.
Une conséquence de cette implémentation est déjà perceptible: cette fonctionnalité transversale est étroitement liée au Compte. Changer d'outil de gestion de traces implique qu'il faille effectuer des modifications à chaque référence à log4j dans la classe Compte. Et chaque modification dans la classe Compte risque d'altérer le bon fonctionnement de notre fonctionnalité de base.

```java
package gestion;

import org.apache.log4j.BasicConfigurator;
import org.apache.log4j.Logger;

/**
 * Gestion de Compte
 */
public class Compte {

    Logger logger = null;
    long solde = 0;

    Compte(long soldeInitial) {
        /* Configuration du logger */
        logger = Logger.getLogger(this.getClass());
        BasicConfigurator.configure();
        /**/
        solde = soldeInitial;
    }

    void retrait(long montant) throws Exception {

        if ( (solde-montant) < -1000 ) {
          /* Cas de refus */
            logger.error("Virement refusé.");
            throw new Exception("Solde insuffisant!");
        } else {
            /* Cas de réussite */
            solde= solde - montant;
            logger.info("Virement de "+montant+" Euros effectué ");
        }
    }
}
```

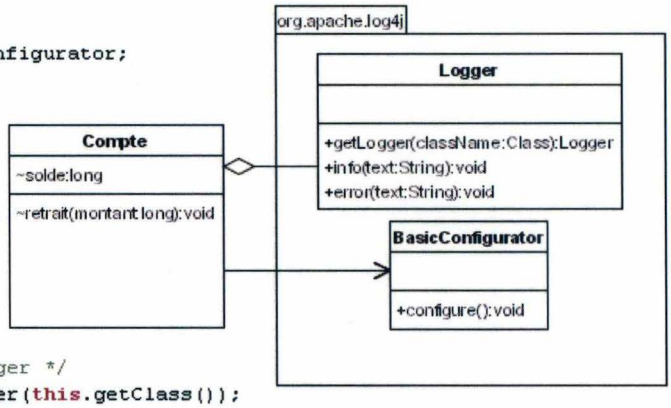



**Figure 3. Classe Compte avec log4j et son diagramme de classes.**

Si l'on généralise ce phénomène à un programme plus complexe constitué de nombreuses classes, force est de constater que du code relatif à la gestion de traces se retrouve dans la plupart des classes qui constituent l'application. On parle dans ce cas de **dispersion de code**. La Figure 4 met en évidence la dispersion de code sur un projet de grande taille. Chaque trait horizontal à l'intérieur des classes correspond à du code lié à la gestion des traces.

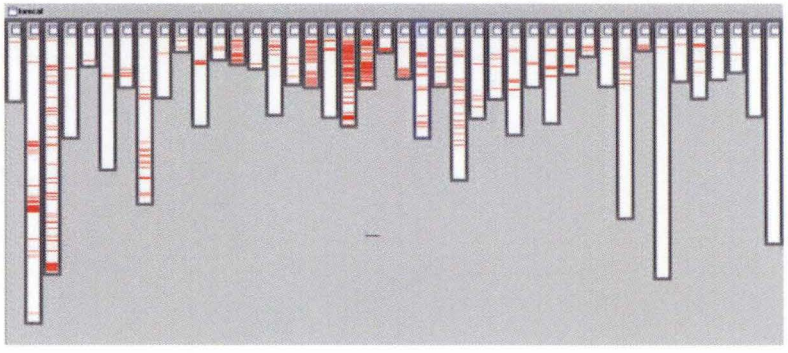


**Figure 4. Gestion des traces dans org.apache.tomcat [Kersten 02]**

Dans cette optique, la limite de modularité du paradigme de programmation orientée objet est clair. Il devient difficile de retirer ou de remplacer un composant ayant une relation aussi forte avec le reste du code. Le nombre de modifications à apporter est trop élevé. Cela remet donc clairement en cause les capacités de maintenance et d'évolutivité de l'application.

Le paradigme de programmation orientée aspect a pour but d'aller plus loin dans la décomposition du programme en isolant les fonctionnalités transversales, nommées de manière plus générale préoccupations transversales, en regroupant leur code dispersé en **aspects**. Cette action peut être vue comme une sorte de factorisation du code, où les appels redondants ne sont plus écrits qu'une fois.

Cependant, il est important de remarquer qu'il ne s'agit pas du seul moyen d'améliorer la modularité du code objet. Ce sujet est abordé succinctement en Annexe 1.

# Section 2. L'objet enrichi par l'aspect

Voici la définition de l'aspect selon [Czarnecki 97]: "Un aspect est une spécification partielle d'un concept ou d'un ensemble de concepts relatif à une préoccupation". L'aspect est donc l'implémentation de la ou des fonctionnalités qui permettent de répondre à une préoccupation considérée.

Dans la pratique, on centralise l'entièreté du code relatif à une ou plusieurs préoccupations transversales dans une seule entité: l'aspect. Avec cette architecture, il devient possible de modifier ou même de soustraire la/les préoccupations du reste de l'application sans avoir à intervenir sur les autres fonctionnalités. Le risque d'introduire des erreurs sur la préoccupation première est réduit au minimum et le gain en capacité de modularité est sans équivoque.
En utilisant les aspects, la classe Compte reste inchangée (cf. Figure 2), son code source est totalement indépendant de la gestion de traces tandis que l'aspect contient tout le code relatif à la fonctionnalité transversale de gestion de traces. Le diagramme de classes de la Figure 5 illustre bien ce fait.

Remarque:
Le stéréotype <<woven class>> signifiant "classe tissée" caractérise une classe liée à un aspect (cf. Section 3), lui-même désigné par le stéréotype <<aspect>>. Ce choix de représentation [Suzuki et al. 99] n'est pas le résultat d'une normalisation. Il en existe d'autres.

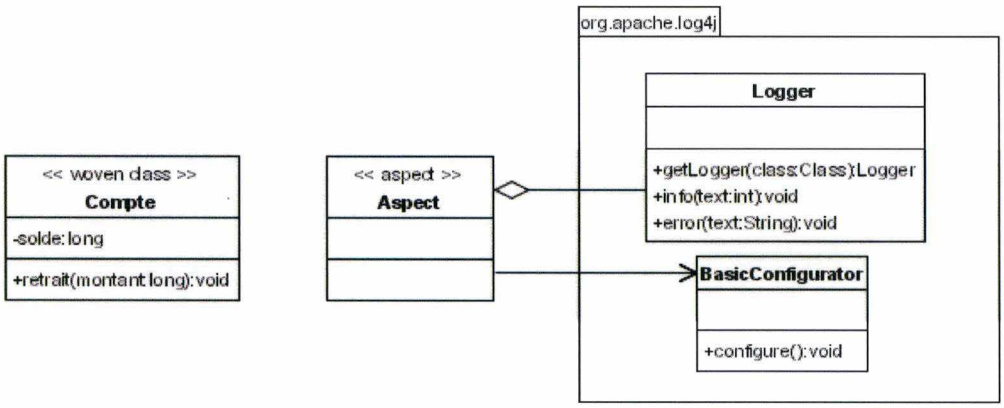



**Figure 5. Diagramme de classes de Compte et son aspect de gestion de traces.**

On identifie principalement 2 types d'éléments dans un aspect. L'un pour définir à quel endroit agir dans l'application, et l'autre pour décrire l'action qui y sera liée:

- Une expression qui décrit un point déterminé dans le flux d'exécution du programme est un **joinpoint**. Par extension, une expression qui regroupe un ensemble de joinpoints est un **pointcut**.
- Le code définissant un ensemble d'opérations à exécuter est appelé **advice**. Il est analogue à une méthode. Il décrit également la nature de la relation entre le code à exécuter et le pointcut (i.e. si le code doit être exécuté avant ou après le pointcut).

La Figure 6 présente l'aspect de gestion de traces conforme à la syntaxe de l'outil AspectJ. La classe Compte est toujours celle présentée en Figure 2, apparemment indépendante, exempte de tout appel à log4j. Les pointcuts déterminent des endroits du code de la classe Compte. Les advices,

reconnaissables par les mots-clefs 'before' ou 'after', définissent quelles actions effectuer sur quel pointcut.
Tout le code relatif à la gestion de traces est centralisé dans l'aspect. La fonctionnalité de gestion de traces n'est plus mélangée à la fonctionnalité de base de gestion des comptes.

```
package gestion;

import org.apache.log4j.Logger;
import org.apache.log4j.BasicConfigurator;

/**
 * Logger pour Compte
 */
public aspect Aspect {

    Logger logger = null;

    pointcut surConstructeur()        : execution(Compte.new(long));
    pointcut surRetrait(long montant): execution(void Compte.retrait(long)) && args(montant);
    pointcut surException()           : handler(Exception);

    before() : surConstructeur() {
        logger = Logger.getLogger(this.getClass());
        BasicConfigurator.configure();
    }

    after(long montant) returning: surRetrait(montant) {
        logger.info("Virement de "+montant+" Euros effectué ");
    }

    before() : surException() {
        logger.error("Virement refusé.");
    }
}
```

**Figure 6. Aspect de gestion de traces pour la classe Compte.**

Il existe plusieurs mécanismes qui s'ajoutent aux notions de bases de l'aspect.
Par exemple, **la déclaration inter-type** d'AspectJ [AspectJ 05] complète le code de la classe instrumentée par l'aspect. On peut de cette manière attribuer de nouveaux champs ou de nouvelles méthodes aux classes souhaitées, tout en gardant dissociés le code source des fonctionnalités transversales par rapport à la fonctionnalité de base. Dans la même lignée, **l'introduction** de JBoss-AOP [JBoss 05] permet de forcer une classe à implémenter une interface.

Tous ces mécanismes dont beaucoup n'ont pas encore été cités enrichissent les notions de l'aspect. Mais ceux-ci diffèrent d'une implémentation à l'autre. En effet, cette matière est encore en cours de standardisation, principalement par l'AOP-Alliance (http://aopalliance.sourceforge.net/).

# Section 3. Les techniques d'intégration de l'aspect

Un tisseur d'aspect, ou **weaver**, est le logiciel qui réalise l'opération d'intégration entre un ensemble de classes et un ensemble d'aspects [Pawlak et al. 04]. Le tissage consiste à établir les liaisons entre classes et advices à partir des pointcuts définis dans les aspects.
Voici les points principaux qui caractérisent l'utilisation et le fonctionnement de tout outil de tissage.

## 3.1. L'aspect

L'aspect doit être écrit dans un langage propre à l'outil utilisé ou écrit en Java.

Dans le deuxième cas, le code à exécuter (l'advice) est programmé dans une méthode de classe, tandis que les pointcuts sont écrits soit en Java, soit sous forme d'annotations ou encore dans des fichiers de configuration en xml.

## 3.2. Le tissage

Généralement, les weavers permettent d'effectuer le tissage de l'application à différentes étapes:

- durant la compilation du code source,
- après compilation du code source en bytecode,
- au chargement des classes par la JVM, à l'aide d'un classloader spécialisé.

Le tissage est qualifié de dynamique si le tisseur est capable de modifier le tissage de l'application durant son exécution. On peut alors changer ou retirer des aspects sans interrompre le fonctionnement de l'application. Sinon on parle de tissage statique.

## 3.3. L'implémentation du tisseur

Les implémentations du paradigme aspect et le fonctionnement des tisseurs varient fort d'un outil à l'autre, raison pour laquelle leurs caractéristiques de fonctionnement sont abordées au cas par cas dans la Section 4 sur l'évaluation des outils AOP.

# Section 4. Étude des outils AOP

## 4.1. Critères de sélection

L'AOSD est une organisation planifiant des conférences pour la promotion des logiciels orientés aspect.
Son site [AOSD 05] met à la disposition du public la liste des outils AOP recensés et considérés comme étant appropriés pour un développement destiné à une mise en production.
Cette liste est le point de départ pour la sélection de l'outil utilisé pour le projet:

AspectJ, AspectWerkz, JBoss-AOP, Spring-AOP, abc, Aspect#, AspectC++, AspectDNG, AspectXML, FeatureC++, JAC, LOOM.NET, Nanning, XWeaver et Eos.

Un premier filtrage peut être effectué à partir des critères dictés par les besoins du projet:

- **L'environnement:**
  Java est le langage utilisé pour la majeure partie du projet. La version utilisée est la 1.4.2, mais le passage à la version 1.5 doit être possible.
- **La capacité d'intégration**:
  L'application BNM est à déployer dans JBoss. L'outil ne doit donc pas causer de problème avec l'utilisation de JBoss version 4.
- **Le coût:**
  La politique de la société est d'utiliser des outils Open Source pour limiter les coûts de licences.
- **Le support**:
  Pour assurer la continuité du projet, il est important que l'outil Open Source ait un support suffisant pour assurer l'apport des corrections et la longévité du produit utilisé.
- **La version:**
  Pour assurer une stabilité suffisante, seules les versions supérieures ou égales à 1.0 sont prises en compte en Final Release ou éventuellement Release Candidate (RC).
- **Le développement:**
  L'IDE Eclipse est utilisé par les développeurs de Micro Research. Cet IDE a la particularité d'être un réel framework d'application. En effet, il est possible d'y greffer un grand nombre de

fonctionnalités sous forme de plug-ins. Une solution AOP qui propose un plug-in pour Eclipse est considérée comme un plus.

Voici les informations relatives aux critères de sélection cités:

| | Environnement | cap. d'intégration | coût | support | version | développement |
|---|---|---|---|---|---|---|
| **AspectJ** | Java | possible | gratuit | certain | 1.2.1 | plug-in Eclipse. |
| **AspectWerkz** | Java | possible | gratuit | incertain | 2.0 | plug-in Eclipse. |
| **JBoss-AOP** | Java | possible | gratuit | certain | 1.1.2 | plug-in Eclipse. |
| **Spring-AOP** | Java | possible | gratuit | certain | 1.2.1 | plug-in Eclipse. |
| **abc** | Java | possible | gratuit | incertain | 1.0.2 | non |
| **Aspect#** | .Net | impossible | gratuit | incertain | 2.1.0 | non |
| **AspectC++** | C++ | impossible | gratuit | incertain | 0.9.3 | plug-in Eclipse. |
| **AspectDNG** | .Net | impossible | gratuit | incertain | 0.6.4 | non |
| **AspectXML** | XML | impossible | gratuit | incertain | 0.8 | non |
| **FeatureC++** | C++ | impossible | gratuit | incertain | 0.1 | non |
| **JAC** | Java | impossible | gratuit | incertain | 0.12.1 | non |
| **LOOM.NET** | .Net | impossible | gratuit | incertain | 1.2 | non |
| **Nanning** | Java | possible | gratuit | incertain | 0.9 | non |
| **XWeaver** | C/C++ | impossible | gratuit | incertain | 0.9.1 | non |
| **Eos** | .Net | impossible | gratuit | incertain | 0.3.2 | non |

La liste des outils disponibles diminue en conséquence:

- Les seuls weavers pour Java sont AspectJ, AspectWerkz, JBoss-AOP, Spring-AOP, abc, JAC et Nanning. Ce sont les seuls susceptibles d'être intégrés au projet.
- Une exception est faite pour JAC qui, malgré le langage Java, est un framework d'application qui ne propose pas de weaver capable de tourner ailleurs que sur sa propre plateforme. De plus JAC n'a pas encore atteint la version 1.0 requise.
- AspectJ, AspectWerkz, JBoss-AOP et Spring-AOP sont classés par le site de l'AOSD comme ayant un nombre d'utilisateurs significatif, ce qui est une assurance de support dans le monde de l'open source.
- Il faut toutefois classer AspectWerkz dans la catégorie des incertains pour le support. En effet, les projets AspectWerkz et AspectJ fusionnent au profit d'AspectJ.

**AspectJ**, **JBoss-AOP** et **Spring-AOP** répondent à tous les critères énoncés précédemment. Ces 3 solutions fournissent également un plug-in pour Eclipse.

| | Environnement | cap. d'intégration | coût | support | version | développement |
|---|---|---|---|---|---|---|
| **AspectJ** | Java | possible | gratuit | certain | 1.2.1 | plug-in Eclipse. |
| **JBoss-AOP** | Java | possible | gratuit | certain | 1.1.2 | plug-in Eclipse. |
| **Spring-AOP** | Java | possible | gratuit | certain | 1.2.1 | plug-in Eclipse. |

Il reste maintenant à évaluer les 3 challengers.

## 4.2. Le comparatif

Le but de cette partie n'est pas d'analyser chaque outil dans ses moindres détails, mais uniquement de mettre en évidence les caractéristiques qui les différencient pour faciliter leur évaluation. Il ne s'agit donc nullement d'une documentation exhaustive. C'est pourquoi le lecteur est invité à visiter les sites officiels de ces produits où il trouvera une documentation détaillée [AspectJ 05] [JBoss 05] [Spring 05].

Les aspects illustrant la présentation des 3 outils sont écrits pour être tissés à la classe ClasseSimple suivante:

```
1  package mem.ex2;
2
3  import java.util.Vector;
4
5  public class ClasseSimple {
6
7      String texte = "Hello World";
8      Vector vecteur = new Vector();
9
10     void helloWorld() {
11         System.out.println(texte);
12     }
13 }
```

**Figure 7. ClasseSimple.java.**

## 4.3. AspectJ

### LE FONCTIONNEMENT

La philosophie d'AspectJ est d'utiliser une extension du langage Java pour la déclaration des aspects et d'effectuer le tissage statiquement.
Le compilateur ajc se charge de compiler le code Java et les aspects, ainsi que du tissage. Le code final obtenu est du bytecode java directement exécutable par la machine virtuelle.

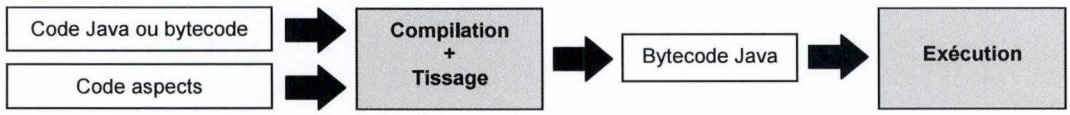



**Figure 8. AspectJ: le weaver lie le code et l'aspect à la compilation, avant l'exécution.**

Dans les cas triviaux, les modifications effectuées par le compilateur ajc sont les suivantes:

- L'aspect devient une classe.
- Les advices de l'aspect deviennent des méthodes de la classe.
- Pour chaque élément de code de l'application qui est visé par un pointcut, le compilateur insère l'appel à l'advice correspondant.

### LE LANGAGE

L'aspect **`AspectSimple.aj`** présenté en Figure 9 modifie **`ClasseSimple.java`** en Figure 7.

```
1  package mem.aspect;
2
3  import mem.ex2.ClasseSimple;
4
5  public aspect AspectSimple {
6
7      /* Déclarations inter-types */
8      private String ClasseSimple.nouvelleVariable = "Déclaration inter-type";
9
10     public void ClasseSimple.nouvelleMethode() {
11         System.out.println("Ceci est une autre "+nouvelleVariable);
12     }
13
14
15     /* Déclarations de pointcuts */
16     pointcut surMethod(): execution(void ClasseSimple.helloWorld());
17     pointcut surMethod2(ClasseSimple cs): target(cs) && execution(void helloWorld());
18
19     /* Déclaration des advices */
20     before() : surMethod() {
21         System.out.println("Avant Hello World.");
22     }
23
24     after(ClasseSimple cs) : surMethod2(cs) {
25         System.out.println("Après Hello World.");
26         cs.nouvelleMethode();
27     }
28 }
```

**Figure 9. AspectJ: le code source de l'aspect AspectSimple.aj.**

### *L'aspect*

L'aspect est déclaré dans un fichier, d'une manière similaire à une classe. On y trouve les pointcuts, les advices et les déclarations inter-types.

Au niveau de la syntaxe:

- `aspect A extends B implements I, J { … }`
  Tout comme les classes, un aspect peut hériter des propriétés d'un aspect abstrait ou implémenter une interface.
- `aspect A percflow( call(void Foo.m()) ) { … }`
  Par défaut, un aspect est un singleton. Cependant, Il est possible de changer cette configuration avec les opérateurs `pertarget(pointcut)`, `perthis(pointcut)`, `percflow(pointcut)`, `percflowbelow(pointcut)`.
  Par exemple, avec `pertarget(pointcut)`, une instance d'aspect est créé par instance ciblées par le pointcut.

### *Le pointcut*

Les pointcuts (lignes 16 et 17 du code de la Figure 9) sont nommés tels une méthode: un mnémonique "pointcut" suivi d'éventuels paramètres entre parenthèses. Tout paramètre est une instance pointée par le pointcut qui peut être récupérée et utilisée dans un advice (lignes 20 à 22 et 24 à 27).
Les expressions font références à des invocations de méthodes, des constructeurs, des déclenchements d'exceptions, des assignations, des accès à des variables, …
Des caractères spéciaux (des "wildcards") permettent de généraliser les joinpoints visés:

- `public String ClasseSimple.helloWorld()` pour une méthode publique sans paramètre qui retourne un String.
- `public * ClasseSimple.helloWorld(..)` pour toute méthode publique ayant n'importe quel(s) paramètre(s) qui retourne n'importe quel type d'objet.

De nombreux opérateurs peuvent être utilisés pour définir les pointcuts. En voici quelques exemples:

- `execute(expr)` définit le poincut aux endroits désignés par l'expression passée en paramètre.
- `call(expr)` définit le poincut aux endroits où se situent l'appel au code désigné par l'expression.
- `handler(type_d'Exception)` cible un type d'exception.
- `cflow(pointcut)` extrait dynamiquement tous les joinpoints survenant durant l'exécution de chaque élément de code référencé par le pointcut passé en paramètre.
- La composition de pointcuts par les opérateurs booléens `||`, `&&` et `!`
- ...

### *L'advice*

Le corps des advices (en lignes 20 à 26 du code de la Figure 9) est écrit en code Java classique, enrichi de certains mots-clés faisant référence aux joinpoints sur lesquels ils agissent.
Par exemple, thisjoinpoint est une référence à l'objet sur lequel l'advice est exécuté.

### *La déclaration inter-type*

Dans l'aspect, il est possible de déclarer des champs et des méthodes d'une classe qui ont trait à une fonctionnalité transversale, propre à l'aspect (en lignes 8 à 12 du code de la Figure 9). Cependant, l'accès par encapsulation (private, package, public) du champ ou la méthode créée est fonction de l'aspect et non de la classe.
Par exemple, la ligne 8 de l'exemple, est

```
private String ClasseSimple.nouvelleVariable = "Déclaration inter-type";
```

Cela signifie que nouvelleVariable est une variable privée de `ClasseSimple`, mais uniquement visible par l'aspect l'ayant déclaré.

## PLUG-IN ECLIPSE

Le plug-in utilisé est AJDT V1.2.0.
Il est très complet, apporte réellement des facilités au programmeur:

- La création d'un projet de type AspectJ est disponible. Ce type de projet met automatiquement en place les librairies nécessaires à la programmation avec AspectJ.
- Tout comme pour le code Java, le code est recompilé avec ajc d'AspectJ à chaque fois qu'il est sauvegardé. Si une erreur de compilation des aspects est détectée, elle est signalée.
- Une vue nommée "Cross References" montre le tissage effectué à la compilation.
    - Si l'on édite `ClasseSimple.java`, on visualise toutes les modifications portées par les aspects à la classe. La fenêtre est présentée à la Figure 10, à gauche.
    - Si l'on édite `AspectSimple.aj`, on visualise toutes les modifications faites par l'aspect. La fenêtre est présentée à la Figure 10, à droite.

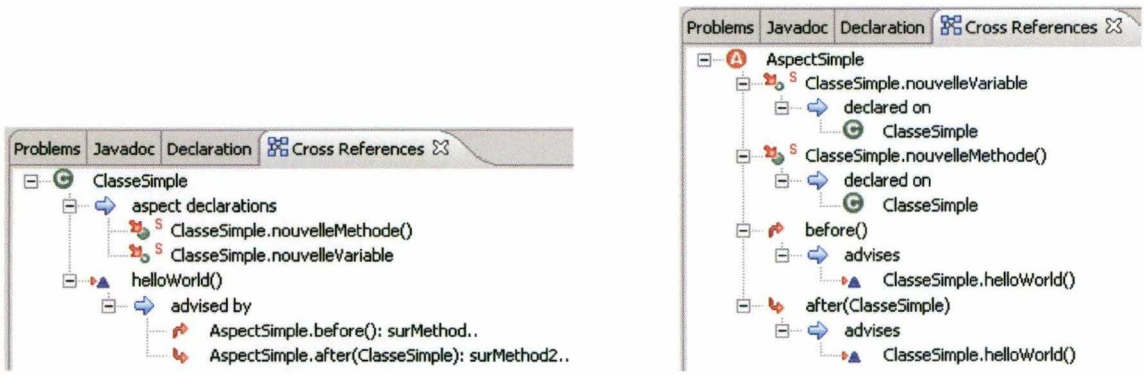



**Figure 10. AspectJ: le résultat du tissage visualisable via le plug-in Eclipse.**

- Lors de l'édition d'une classe Java, tout joinpoint situé dans la classe est mis en évidence par une flèche située à gauche de la ligne de code correspondant au joinpoint. Le programmeur est donc visuellement alerté qu'un code intermédiaire (i.e. l'advice) est exécuté à cet endroit précis.

- Faisant plus état de démonstration technologique que d'utilité avérée, il est possible d'obtenir une visualisation graphique des classes dans lesquelles les joinpoints sont mis en évidence. Les classes sont présentées sous formes de bâtonnets, comme dans la Figure 4 en page 5.

### PERFORMANCES

Effectuer un comparatif de performance entre les différents outils s'avère être une tâche difficile, tant le nombre de caractéristiques pouvant être testées est grand et les moyens d'optimisations nombreux. C'est la raison pour laquelle ce travail se base sur le benchmark effectué par des spécialistes en la matière [AspectWerkz Bench 04], ce qui suffira à donner une notion d'ordre de grandeur entre les outils évalués.

Le benchmark met en évidence l'avantage manifeste du tissage statique. Le tissage à l'étape de compilation permet d'obtenir un bytecode performant. C'est de loin l'outil qui offre les meilleures performances lors des tests à l'exécution de l'application de référence.

### POINTS FORTS / POINTS FAIBLES

***Points forts***

- La richesse du langage [AspectJ 05 a] permet de déclarer des aspects précis et complexes.
- La déclaration inter-type est attrayante car proche d'une déclaration en Java.
- Le tissage statique permet d'identifier des problèmes à la compilation, avant d'exécuter l'application.
- Les pertes en performances sont limitées.
- Le plug-in Eclipse est le plus abouti et très efficace.

***Points faibles***

- L'enrichissement du langage Java peut être vu comme un surcroît de complexité.
- Le tissage statique impose de recompiler l'application à chaque modification. La recompilation nécessite de disposer de toutes les librairies nécessaires, ce qui n'est pas toujours évident si l'application est un produit étranger à la société [Bodkin 05].
- ajc compile l'application, ses librairies et ses aspects à sa seule exécution. Les besoins en ressources systèmes pour effectuer cette tâche peuvent être conséquents.
- Le tissage statique ne permet aucun changement au niveau des aspects durant l'exécution de l'application.

### REMARQUES

À ce jour (le 01/06/2005), la nouvelle version d'AspectJ 1.5.0 est au stade de développement (dont la dernière version est M2 pour "milestone2"). Celle-ci autorise la déclaration d'aspects en Java et l'exécution du tissage durant la phase de chargement des classes dans la JVM par le ClassLoader. Il ne s'agit pourtant pas encore de tissage dynamique car il n'est exécuté qu'à la condition que tous les aspects impliqués soient chargés avant la classe de l'application [AspectJ 05 b].
Il n'est donc toujours pas possible d'installer une nouvelle version d'un aspect sans redémarrer toute l'application. Par contre, il ne faut plus disposer que des librairies utilisées au run-time.

## 4.4. JBoss-AOP

### LE FONCTIONNEMENT

JBoss-AOP fait partie de la catégorie des tisseurs dynamiques et se base sur un design pattern déjà fort usité dans l'architecture de JBoss: les intercepteurs. Le principe est d'exécuter une chaîne

d'intercepteurs[2] avant que le code de l'élément appelé ne soit exécuté. Chaque intercepteur est une classe dérivant du type `Interceptor`, contenant une méthode correspondant à un advice.

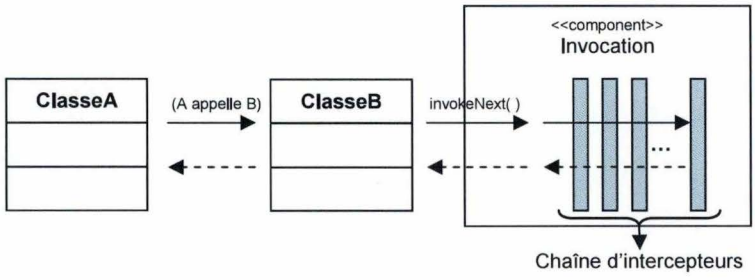



**Figure 11. Le principe de la chaîne d'intercepteurs sous JBoss-AOP.**

Avant que l'application ne puisse être tissée, JBoss-AOP nécessite une étape de compilation du bytecode supplémentaire: **l'instrumentation**.

- L'instrumentation insère le code nécessaire aux endroits définis par les pointcuts pour que l'application puisse appeler la méthode `invokeNext()` d'un objet de type Invocation. Le remaniement du code est statique. Il est effectué soit via compilateur avant l'exécution (c'est le cas illustré par la Figure 12), soit automatiquement lors du chargement des classes par la machine virtuelle (JVM) durant l'exécution.
- Le tissage des aspects s'effectue en ajoutant la référence des intercepteurs (advices) dans la chaîne d'intercepteurs de l'objet Invocation correspondant aux joinpoints déclarés. Le tissage est dynamique: il est effectué lors du chargement des "classes-aspects" par un classloader spécialisé.

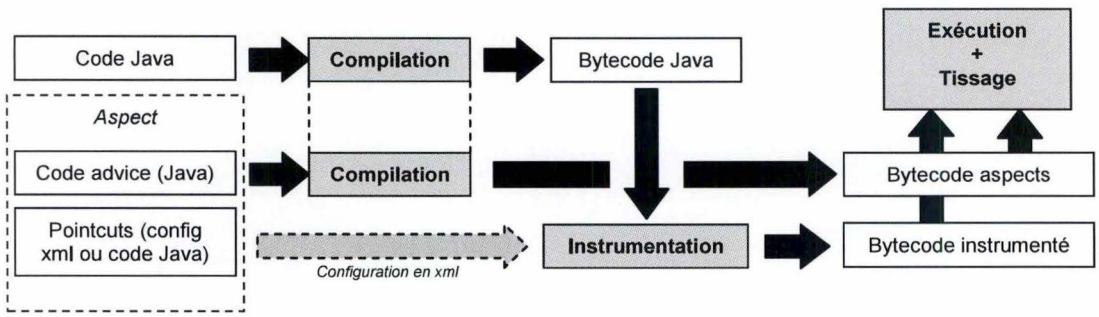



**Figure 12. Tissage avec JBoss-AOP: le tissage est effectué à l'exécution.**

## LE LANGAGE

### *L'aspect*

Les aspects sont divisés en 2 parties. Les advices sont écrits en Java dans une classe, tandis que le reste est déclaré dans un fichier xml ou sous forme d'annotation dans la même classe.

[2] La chaîne d'intercepteurs (interceptor chain) est aussi appelée pile d'intercepteurs (interceptor stack).

```
1  package mem.aspect;
2
3  import org.jboss.aop.joinpoint.MethodInvocation;
4
5  public class AspectSimple {
6
7      public Object surMethod(MethodInvocation invocation) throws Throwable {
8          try {
9              System.out.println("Avant Hello World.");
10             return invocation.invokeNext();
11         } finally {
12             // rien
13         }
14
15     }
16
17     public Object surMethod2(MethodInvocation invocation) throws Throwable {
18         try {
19             return invocation.invokeNext();
20         } finally {
21             System.out.println("Après Hello World.");
22         }
23     }
24 }
```

**Figure 13. JBoss-AOP: les advices d'un aspect dans une classe Java.**

Le même exemple que celui présenté pour AspectJ a été repris pour JBoss-AOP. Le code de `ClasseSimple.java` en Figure 7 reste inchangé tandis que l'aspect AspectSimple est présenté sous forme de combinaison Java/xml en Figure 13 et Figure 14, et en Java avec des annotations en Figure 15.

### *Le fichier xml*

Il décrit toutes les caractéristiques de l'aspect, hormis le code des advices.

```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <aop>
3     <aspect class="mem.aspect.AspectSimple" scope="PER_VM"/>
4     <bind pointcut="execution( void mem.ex2.ClasseSimple-&gt;helloWorld())">
5         <advice aspect="mem.aspect.AspectSimple" name="surMethod"/>
6         <advice aspect="mem.aspect.AspectSimple" name="surMethod2"/>
7     </bind>
8 </aop>
```

**Figure 14. JBoss-AOP: la configuration de l'aspect dans un fichier xml.**

Dans l'exemple en Figure 14, on y voit:

- en ligne 3 la déclaration de l'aspect et de son champ d'action
    - `PER_VM`: pour que l'aspect soit unique,
    - `PER_CLASS`: pour qu'un aspect soit instancié par classe interceptée,
    - `PER_INSTANCE`: pour qu'un aspect soit instancié par instance interceptée.
- en ligne 4 à 7 le tissage où l'on associe pointcut et advice.

D'autres éléments venant enrichir l'aspect sont possibles comme:

- Le `cflow` pour définir un pointcut selon la pile des appels. Par exemple A est un joinpoint uniquement s'il a été appelé par B (même s'il y a eu des appels intermédiaires).
- Le `dynamic-cflow` permet de déterminer si un joinpoint doit être exécuté ou non, selon un test dynamique écrit dans une méthode en Java.
- l'`introduction` pour forcer une classe à implémenter une interface et le `mixin` pour désigner quelle classe fournit l'implémentation des méthodes de l'interface introduite. Le mécanisme d'`introduction/mixin` peut-être considéré comme l'équivalent de la déclaration inter-type.
- ...

### *Les annotations*

Contrairement à l'utilisation d'un fichier xml, les annotations permettent de conserver toute la déclaration d'un aspect dans le seul fichier de la classe. Cependant, les annotations entre Java 1.4.2 et 1.5 sont différentes.

Dans le cas de la 1.4.2, et comme on le voit à la Figure 15, les annotations sont inscrites à l'intérieur des commentaires Javadoc dont le code est généré par une étape supplémentaire de compilation. Il y a au final 3 étapes de compilation: la compilation du code source avec javac, la compilation des annotations avec annotationc, et finalement l'instrumentation avec aopc (ou au chargement de la classe).

Dans le cas de Java 1.5, les annotations font partie intégrante du langage et sont directement insérées dans le code. Il n'y a donc pas d'étape de compilation supplémentaire que celle de l'instrumentation.

```
1  package mem.aspect;
2
3  import org.jboss.aop.joinpoint.MethodInvocation;
4  import org.jboss.aop.pointcut.Pointcut;
5
6  /**
7   * @@org.jboss.aop.Aspect (scope = Scope.PER_VM)
8   */
9  public class AspectSimpleAnnotation {
10
11     /**
12     * @@org.jboss.aop.PointcutDef ("execution(void mem.ex2.ClasseSimple->helloWorld())")
13     */
14     public static Pointcut poincutsurMethod;
15
16     /**
17     * @@org.jboss.aop.Bind (pointcut="mem.aspect.AspectSimpleAnnotation.poincutsurMethod")
18     */
19     public Object surMethod(MethodInvocation invocation) throws Throwable {
20         try {
21             System.out.println("Avant Hello World.");
22             return invocation.invokeNext();
23         } finally {
24             // rien
25         }
26     }
27
28     /**
29     * @@org.jboss.aop.Bind (pointcut="mem.aspect.AspectSimpleAnnotation.poincutsurMethod")
30     */
31     public Object surMethod2(MethodInvocation invocation) throws Throwable {
32         try {
33             return invocation.invokeNext();
34         } finally {
35             System.out.println("Après Hello World.");
36         }
37     }
38 }
```

**Figure 15. JBoss-AOP: classe aspect avec annotations pour Java1.4.2**

### *Le pointcut*

Le pointcut est décrit dans un fichier xml ou via le système des annotations. Dans les 2 alternatives le pointcut est définit par le mot `pointcut` et l'expression qui le spécifie. Comme dans AspectJ, des caractères spéciaux peuvent généraliser le pointcut à un groupe de packages, de classes, de variables, de méthodes, …

Des opérateurs viennent préciser le pointcut:

- `execution(expr)` définit le poincut aux endroits désignés par l'expression,
- `call(expr)` définit le poincut à tous les endroits qui font appel au code ciblé par l'expression.

- `$instanceof {class}` désigne toute classe dérivant de la classe spécifiée entre parenthèses.
- La composition de pointcuts par les opérateurs booléens **AND**, **OR** et **!**

### *L'advice*

Un advice est une méthode déclarée dans un intercepteur, c'est-à-dire une classe qui implémente l'interface `org.jboss.aop.advice.Interceptor`. La méthode doit être déclarée comme suit:
`public Object invoke(Invocation invocation)throws Throwable.`
De ce fait, seul un advice est susceptible d'être déclaré par intercepteur. Cependant, une facilité a été apportée. Comme dans les Figure 13 et Figure 15, il est permis de déclarer plusieurs advices dans une seule classe avec des noms de méthodes distincts.
Côté syntaxe, il faut remarquer l'appel `invocation.invokeNext();`
Cette opération consiste à appeler l'intercepteur suivant dans la chaîne d'intercepteur. S'il s'agissait du dernier intercepteur, c'est le code initial intercepté par l'aspect qui est invoqué.

## PLUG-IN ECLIPSE

Le plug-in utilisé est JBoss AOP IDE version 1.0.1
Il est peu mature, et n'offre encore que le strict minimum. Mais le monde de JBoss étant très actif, on peut s'attendre à avoir une version plus fournie dans un avenir assez proche.
Les facilités existantes sont:

- La création d'un projet de type JBoss-AOP est disponible. Ce type de projet met automatiquement en place les librairies nécessaires à la progammation avec JBoss-AOP.
- L'ajout automatique d'un advice, d'un aspect, d'un intercepteur ou d'un pointcut dans le fichier de configuration xml via la fenêtre "AspectManager". Cette option est intéressante lorsque l'on n'est pas encore familiarisé avec la syntaxe des expressions définissant les pointcuts. Par contre, il faut écrire soit même les expressions dès que l'on aborde des cas non triviaux ou que l'on n'utilise pas le fichier xml.
- La fenêtre "AdvisedMember" présente les éléments de la classe en cours d'édition qui vont être tissés. Pour une raison restée inconnue, cette fonctionnalité a connu des problèmes sur les deux machines sur lequel le travail de développement a été effectué, la fenêtre restant désespérément vide…

Remarque:
La web-console du serveur d'applications JBoss a le mérite de combler partiellement les lacunes du plug-in. Si l'on déploie une application dans JBoss, il est possible via un browser Internet de visualiser le résultat du tissage dans l'ensemble des classes déployées (voir Figure 16).

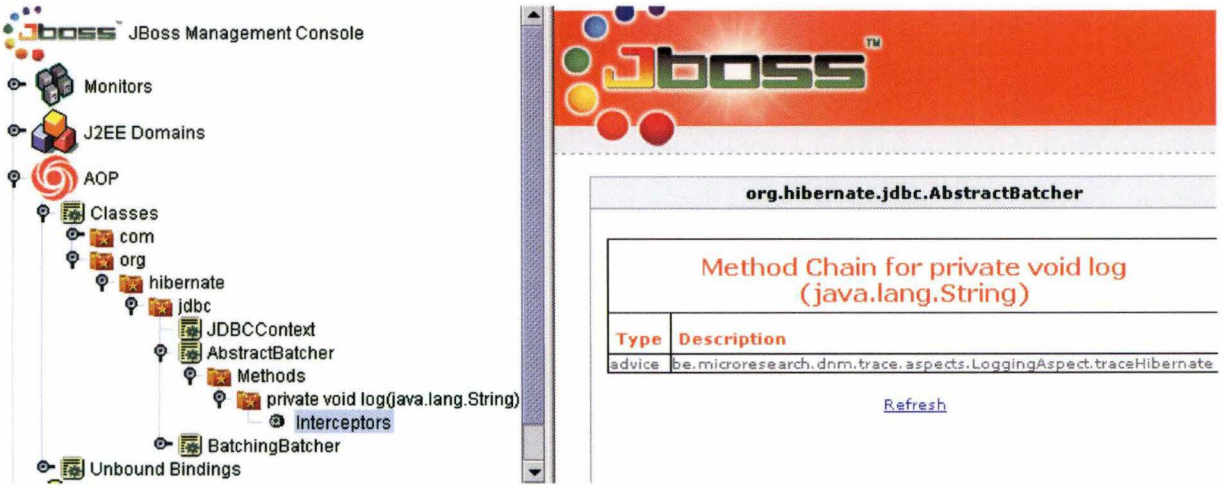



**Figure 16. La web-console de JBoss affiche le tissage d'une méthode d'Hibernate.**

### PERFORMANCES

La complexité imposée pour l'implémentation du design pattern des intercepteurs a un impact non négligeable. Le niveau de performance est clairement inférieur à AspectJ [AspectWerkz Bench 04].

Il faut toutefois noter que le groupe JBoss utilise déjà les chaîne d'intercepteurs à plusieurs niveaux dans son serveur d'applications, notamment pour implémenter les spécification des EJB's en version 3. Visiblement, ils considèrent que les pertes de performance sont suffisamment limitées car elles sont principalement engendrées par le nombre important d'appels à des méthodes intermédiaires. Ceci engendre une augmentation linéaire de la charge, proportionnellement à la quantité d'aspects tissés.

### POINTS FORTS / POINTS FAIBLES

***Points forts***

- La possibilité d'ajouter, de modifier ou de retirer les aspects durant l'exécution, sans entraver le bon fonctionnement de l'application.
- Le langage de programmation est du Java classique.
- Mise à disposition de librairies d'aspects fournissant des services "prêt à l'emploi" comme JBoss-Cache.
- La web-console de JBoss (voir Figure 16) apporte des informations sur les liaisons dynamique entres classes et aspects. Mais elle n'est utilisable que si l'application est déployée sur le serveur d'applications JBoss.

***Points faibles***

- L'absence de compilation spécifique empêche de détecter des erreurs dans le fichier xml, mais également de valider le tissage des aspects avant le démarrage de l'application.
- L'obligation d'instrumenter les classes avant de déployer l'application est contraignante.
    - L'oubli d'instrumenter ne génère pas d'erreur. Cela peut générer des pertes de temps lorsque le développeur s'étonne que ses aspects ne fonctionnent pas.
    - Si l'instrumentation doit être étendue à des composant déjà déployés de l'application, il n'y a plus d'autres alternatives que d'instrumenter puis de redéployer l'application.
- Les possibilités de définition d'aspect de JBoss-AOP ne traduisent pas tout ce qu'offre AspectJ avec sa syntaxe élaborée.
- Les performances sont moins bonnes du fait des nombreux appels intermédiaires propres au design pattern intercepteur.
- Le plug-in Eclipse apporte une aide limitée au développeur. Ceci s'explique surtout par la jeunesse de l'outil.

### REMARQUES

JBoss-AOP un composant du framework d'application JBoss. Cependant il peut être utilisé en dehors du serveur d'applications, pour une application isolée.

## 4.5. Spring AOP

### LE FONCTIONNEMENT

Tout comme JBoss-AOP, Spring AOP est le composant d'un framework d'application J2EE et est utilisable en dehors de son serveur d'applications. Son architecture est également très proche; il est basé sur le design pattern des intercepteurs et fournit un tissage dynamique.

Il y a toutefois une différence notable. Spring AOP utilise un proxy dynamique pour insérer la chaîne d'intercepteurs[3]. L'implémentation du proxy est celle de l'API Java de Sun dans le cas où la classe ciblée dispose d'une interface, sinon c'est celle de la librairie CGLIB [CGLIB 04] pour les classes n'en ayant pas.

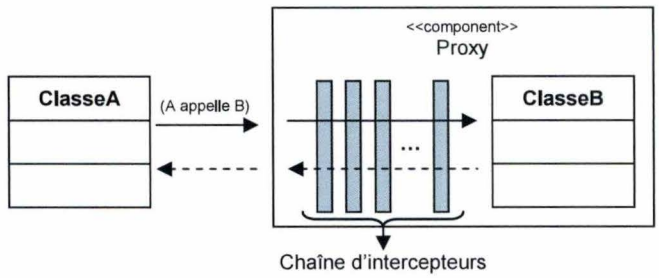



**Figure 17. Proxy et chaîne d'intercepteurs pour Spring AOP.**

Spring AOP est conçu pour être utilisé conjointement avec le composant Spring IoC. IoC est un mécanisme permettant d'injecter les références nécessaires à un objet durant son exécution.
Le framework de Spring permet de déclarer la relation entre préoccupations externes et la classe dans un fichier de configuration xml.

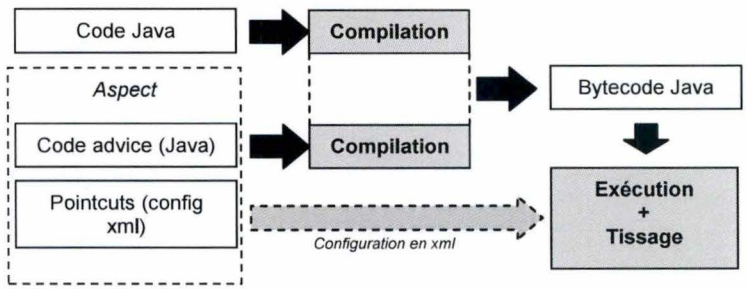



**Figure 18. Tissage avec Spring AOP: il n'y a pas de phase d'instrumentation.**

L'utilisation du proxy évite de devoir passer par une phase d'instrumentation. Par contre, toute classe à tisser doit être chargée par le container de Spring et donc inscrite dans son fichier de configuration. Quand il est utilisé en dehors de son serveur d'applications, le container doit être instancié explicitement dans le code source de l'application.

## LE LANGAGE

### *L'aspect*

Un aspect est déclaré en plusieurs parties. Chaque advice est décrit par une classe Java, tandis que le reste est déclaré dans un fichier xml. Le framework de Spring supporte aussi les annotations de Java1.5, mais ne propose rien pour son utilisation avec Spring AOP.

Remarque:
Pour rappel, le code source de l'aspect présenté découle toujours de la classe `ClasseSimple` dont le code source est présenté en Figure 7, page 10.

[3] Les concepteurs de JBoss-AOP ont également envisagé l'alternative du proxy. Ils ont malgré tout préféré la solution de l'instrumentation comme en témoigne une discussion interne dans leur forum [JBoss 01].

```
1  <beans>
2     <!--CONFIG-->
3     <bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>
4     <bean id="businesslogicbean" class="mem.ex2.ClasseSimple"/>
5
6     <!--ADVISOR-->
7     <bean id="beforeAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
8        <property name="advice">
9          <ref local="beforeAdvice"/>
10       </property>
11       <property name="pattern">
12          <value>.*</value>
13       </property>
14    </bean>
15
16    <bean id="afterAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
17       <property name="advice">
18         <ref local="afterAdvice"/>
19       </property>
20       <property name="pattern">
21          <value>.*</value>
22       </property>
23    </bean>
24
25    <!--ADVICE-->
26    <bean id="beforeAdvice" class="mem.aspect.BeforeAdviceSurMethod"/>
27    <bean id="afterAdvice"  class="mem.aspect.AfterAdviceSurMethod2"/>
28 </beans>
```

**Figure 19. Spring AOP: configuration de l'aspect via le fichier xml.**

### *Le fichier xml*

La philosophie générale de l'outil est que tout élément servant à la configuration de l'aspect (pointcut, advice, advisor, gestionnaire de proxy) est une classe (un **`bean`**) qui doit être configurée dans ce fichier (voir Figure 19).

Toutes les informations nécessaires au tissage de l'application y sont décrites:

- La classe **`ClasseSimple`** (en ligne 4) pour permettre son isolement par le proxy.
- L'**`advisor`** (de la ligne 7 à 14, et 16 à 23) qui a pour rôle de matérialiser l'aspect en associant pointcuts et advice. Dans notre exemple, le **`RegexMethodPointcutAdvisor`** est un advisor qui permet de déclarer les pointcuts à l'aide d'expressions régulières respectant la syntaxe de Perl 5. Il existe d'autre type d'advisors.
  Le **`DefaultPoincutAdvisor`** par exemple, nécessite d'utiliser les identifiants des pointcuts déclarés ailleurs dans le fichier xml.
- La classe de la fabrique de proxy (factory) qui est condigurée en ligne 3.
  Ici, le **`DefaultAdvisorProxyCreator`** a pour charge, à partir des advisors existants, de créer le proxy et de lui fournir les intercepteurs ad hoc.
  Au contraire, le **`ProxyFactoryBean`** nécessite de spécifier la classe ciblée, ses intercepteurs et d'autres paramètres de configuration propre à son fonctionnement tel le type de proxy utilisé, l'interface de la classe si c'est un proxy dynamique, un singleton, ...
- Les advices sont déclarés (en lignes 26 et 27) ainsi que les pointcuts (il n'y en a pas dans cet exemple).

On peut retrouver également dans ce fichier la configuration de différents mécanismes, tel l'introduction et le mixin pour forcer une classe à implémenter une interface ou le controlflow pour identifier si un appel à un joinpoint provient d'une méthode déterminée.

Généralement, pour chaque type d'élément (**`bean`**) à déclarer, il faut joindre une classe qui fournit plus ou moins de facilités parmi ceux disponibles dans la librairie de Spring-AOP. Mais le développeur a aussi la possibilité de développer ses propres classes pour répondre à des besoins plus particuliers.

### *Le pointcut*

Les pointcuts peuvent être utilisés sous forme d'expressions régulières, comme dans notre exemple pour l'advice.
Chaque pointcut peut aussi être déclarés dans le fichier de configuration en spécifiant son nom, son type de classe et les informations nécessaires pour le type de classe choisi.

```
<!-- POINTCUT -->
<bean id="classe_simple_pointcut" class="org.springframework.aop.support.Perl5RegexpMethodPointcut">
 <property name="patterns">
     <value>.*</value>
 </property>
 </bean>
```

**Figure 20. Spring AOP: le pointcut, tel qu'il aurait pu être déclaré pour l'exemple.**

Les types de classes principaux sont basés

- sur les regex (`Perl5RegexpMethodPointcut` et `JdkRegexpMethodPointcut`),
- le nom des méthodes (`NameMatchMethodPointcut`),
- et les controlflows (`ControlFlowPointcut`).

On peut utiliser des pointcuts plus complexes en implémentant soi-même sa classe Pointcut, en utilisant notamment les opérateurs ensemblistes (union, intersection, ...) disponibles via les méthodes statiques de la classe `org.springframework.aop.support.Pointcuts`.

### *L'advice*

L'advice est une classe devant implémenter une de ces 4 interfaces:

- `AroundAdvice` pour le code devant être exécuté avant et après le joinpoint,
- `BeforeAdvice` pour le code devant être exécuté avant le joinpoint uniquement,
- `AfterReturningAdvice` pour le code devant être exécuté après le joinpoint uniquement,
- `ThrowsAdvice` pour le code à exécuter si le joinpoint retourne une exception.

Le code source décrivant l'action à effectuer est écrit dans une méthode dont la signature dépend de l'interface implementée.

```
 1 package mem.aspect;
 2
 3 import java.lang.reflect.Method;
 4 import org.springframework.aop.MethodBeforeAdvice;
 5
 6 public class BeforeAdviceSurMethod implements MethodBeforeAdvice  {
 7
 8     public void before(Method m, Object[] args, Object target) throws Throwable {
 9         System.out.println("Avant Hello World.");
10     }
11 }
```

**Figure 21. Spring AOP: le code source du "BeforeAdvice" de l'exemple.**

```
 1 package mem.aspect;
 2
 3 import java.lang.reflect.Method;
 5
 6 public class AfterAdviceSurMethod2 implements AfterReturningAdvice {
 7
 8     public void afterReturning(Object returnValue, Method m, Object[] args, Object target)
 9     throws Throwable {
10         System.out.println("Après Hello World.");
11     }
12 }
```

**Figure 22. Spring AOP: le code source de l'"AfterAdvice" de l'exemple.**

Les classes des advices sont ensuite utilisées dans le fichier de configuration xml pour spécifier le tissage désiré (lignes 26 et 27 de la Figure 19).

### PLUG-IN ECLIPSE

Le plug-in est prévu pour facilité l'utilisation des fichiers de configuration en xml, notamment pour détecter les erreurs de syntaxe.

Aucune facilité propre à AOP n'est disponible. Le développeur doit se contenter du fichier xml pour évaluer le champ d'action de ses aspects.

### PERFORMANCES

Le benchmark [AspectWerkz Bench 04] montre des résultat similaires (quoique légèrement inférieurs) a JBoss-AOP. L'analogie de l'architecture logicielle explique aisément ce résultat.

AspectJ est donc nettement plus rapide que Spring-AOP et JBoss-AOP.

### POINTS FORTS / POINTS FAIBLES

#### *Points forts*

- Le tissage est dynamique
- Le langage de programmation est du Java classique.
- Il n'y a pas d'instrumentation à prévoir avant de lancer l'application. Il ne faut donc pas prévoir à l'avance les pointcuts qui sont potentiellement utilisables.

#### *Points faibles*

- Pour utiliser Spring AOP en dehors de son serveur d'applications, le code source de l'application doit utiliser l'API de Spring pour instancier les classes à tisser[4] (voir Figure 23). On perd un des principaux avantages de AOP: l'indépendance du code source de l'application.
- Le plug-in très pauvre pour la partie AOP. Il n'y a aucun moyen de visualiser la portée du tissage. Le seul moyen de valider le fichier xml et le tissage est de tester l'application à l'exécution.
- Il n'est pas possible d'intercepter des champs. C'est un choix délibéré des créateurs de Spring pour ne pas violer le principe d'encapsulation de la programmation orientée objet [Spring 05 a]. Même si le point de vue est tout à fait défendable, cela reste néanmoins une limitation par rapport aux 2 autres produits.
- Les performances sont comparables à JBoss-AOP (et donc en deçà d'AspectJ).

[4] À titre de comparaison, pour utiliser JBoss-AOP en dehors de son serveur d'application, il faut spécifier les arguments relatifs au tisseur et sa configuration dans la commande Java servant à instancier la machine virtuelle. Le code source de l'application reste donc inchangé.

```
package mem.ex2;

import org.springframework.context.ApplicationContext;

public class MainApplication
{
   public static void main(String [] args)
   {
      // Instanciation du container
      ApplicationContext ctx = new FileSystemXmlApplicationContext("/conf/springconfig.xml");

      //Instanciation de la classe tissée
      ClasseSimple testObject = (ClasseSimple) ctx.getBean("businesslogicbean");

      //Execute the public method of the bean (the test)
      testObject.helloWorld();
   }
}
```

**Figure 23. Spring AOP: code source de l'application utilisant ClasseSimple.**

## 4.6. Le constat de l'évaluation

JBoss-AOP et Spring AOP ont un mode de fonctionnement assez proche. Entraînant des pertes sensibles en performances mais proposant le tissage dynamique, ils conviennent bien à l'environnement pour lequel ils sont développés: les serveurs d'applications. Ils fournissent le moyen de joindre des services de manière transparente aux applications, même pendant leur exécution. Cependant, Spring AOP n'a pas de moyen de valider un tissage autrement que par des tests à l'exécution. De plus, son utilisation oblige à instancier le container de Spring dans l'application, ayant pour conséquence la création d'un couplage avec la partie aspect, ce que l'on souhaite éviter. À la suite de ces constatations, on peut donc éliminer aisément Spring AOP des outils candidats pour le projet BNM.

AspectJ est le plus performant, il se base sur un langage qui lui est propre, possède le plug-in le plus complet et son tissage est statique. Il permet probablement les tissages les plus complexes entre classes et aspects, tout en assurant les meilleures performances. Il a tous les avantages si ce n'est l'obligation de devoir recompiler l'application avec toutes les sources impliquées, à chaque fois qu'une modification doit être faite pour un aspect.

Il reste donc à trancher entre 2 caractéristiques intéressantes: soit le tissage dynamique avec JBoss-AOP, soit une plus grande garantie de performances avec AspectJ. Ce choix devant être décidé en accord avec les responsables du projets, il convient de le présenter à l'analyse du projet à la Section 3 en page 29.

# Chapitre 2. Analyse et design du projet

## Section 1. Contexte Général

### 1.1. Micro Research S.A.

Depuis sa création en 1986, Micro Research propose des applications spécialisées dans le domaine des télécommunications. Les premières solutions avaient pour tâche de permettre à des applications clientes de communiquer avec des centraux téléphoniques de marques différentes de manière uniforme. Ce type d'outil appelé à l'époque outil de médiation peut être considéré comme un broker.

Avec l'avènement des réseaux de communications de nouvelles générations, les besoins de gestion se sont déplacés sur les équipements de transmission numérique de données. C'est pourquoi Micro Research propose aujourd'hui de nouvelles applications offrant les mêmes facilités pour le monde du data (ex: des routers de réseau à fibres optiques, ...).
Les solutions offertes ont un rôle de couche intermédiaire entre les éléments du réseau et les applications clientes. Le projet MR-BNM (pour "Broadband Network Management") a vu le jour dans le but de ne plus se limiter à un rôle de couche de transmission intermédiaire, mais également d'offrir des services de haut niveau répondant aux besoins des sociétés clientes.

### 1.2. MR-BNM

En plus de facilités de communication avec les éléments du réseau, le projet MR-BNM propose des services de plus haut niveau centrés sur les réseaux IP et DSL. Pour cela, un large éventail de fonctionnalités est développé:

- Un système de détection automatique du réseau où le se système est connecté.
- Le stockage de la topologie du réseau dans une base de données.
- Les connecteurs vers les différents éléments du réseau ou des logiciels de niveau inférieur.
- Un système de synchronisation entre la configuration de la topologie en base de données et la topologie physique.
- La configuration automatique des éléments du réseau.
- La collecte des données provenant du réseau (trafic, performance, erreurs émises, ...).
- L'activation de services personnalisés à destination du client final (bande passante accrue, services audiovisuels, VPN, ...).
- Les différentes interfaces utilisateurs nécessaires aux opérateurs.

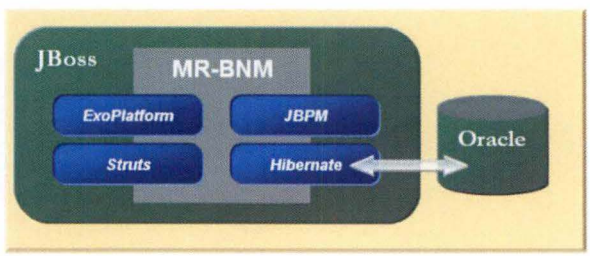



**Figure 24. BNM: Architecture logicielle.**

Le système d'information a une architecture distribuée. L'ensemble du système est construit à partir d'outils Open Source (voir Figure 24):

- JBoss4 est le serveur d'applications J2EE (fournissant principalement un web-server et un container de composants Java) [JBoss 05].
- exo platform1.0 est le système de gestion de portail qui permet la gestion de portlets [eXo platform 04]. Une portlet est une application web enregistrée dans un portail. Le portail permet de personnaliser les accès aux portlets selon les types d'utilisateurs.
- Hibernate3.0 est un outil permettant d'effectuer le mapping objet/relationnel [Hibernate 05]. Les instances Java sont converties au format relationnel et stockées en base de données.
- JBPM2.0 est utilisé pour la prise en charge du workflow [JBPM 05]. Il permet de définir la succession de tâches nécessaires à la réalisation d'un processus.
- Le framework Struts1.2 est utilisé pour garantir l'architecture MVC du système [Struts 05]. Il aide à séparer les composants liés à la visualisation (interfaces graphiques) de ceux du modèle (logique métier) par l'intermédiaire du contrôleur.
- Une base de donnée OracleV9 est utilisée pour la persistance des données [Oracle 05].

## 1.3. Étendue de l'analyse

Elle traite de la partie logging, monitoring et benchmarking à implémenter dans l'application BNM:

- Le logging est aussi appelé gestion des traces. Le but de logging est de suivre le déroulement des requêtes traitées par l'application (cf. définition de requête en page 26). Ce suivi est obtenu en collectant les données générées par les différents composants durant le traitement de chaque requête. L'utilisateur est alors libre d'aller les consulter.
- Le monitoring a pour but de mesurer les performances de l'application. Pour ce faire, la durée de chaque requête traitée par l'application BNM est mesurée. Ces temps d'exécution fournissent au système à développer les moyens de déduire les temps moyens d'exécution selon des critères que la présente analyse doit définir.
- Le benchmarking reprend les mesures déduites du monitoring et les résultats obtenus sur l'application dans un autre environnement (hardware ou software). Le système doit permettre de fournir un comparatif clair entre les mesures des différents monitorings.

L'analyse porte sur l'entièreté du processus d'analyse, à partir de la définition des besoins jusqu'à la conception physique.

## 1.4. Notations

Pour éviter tout risque de confusion, l'application découlant du projet BNM est appelée **application BNM** ou tout simplement **BNM**, tandis que le composant de logging, monitoring et benchmarking analysé et conçu dans ce document est désigné sous le nom de **système**.

Concernant le reste de l'analyse:

- Chaque descriptif du domaine est indexé par la lettre D suivi de son identifiant numérique (D1, D2, D3, ...)
- Chaque objectif est indexé par le terme Obj suivi de son identifiant numérique (Obj1, Obj2, ...). Dans le cas d'un objectif secondaire, l'identifiant de l'objectif père est repris, suivi d'un point, puis d'un nouvel identifiant (Obj3.1, Obj3.2, Obj3.3, ...).
- Chaque exigence est indexée par le terme Ex suivi de son identifiant numérique (Ex1, Ex2, Ex3, ...).
- Chaque spécification est indexée par le terme Spec suivi de son identifiant numérique (Spec1, Spec2, Spec3, ...).

L'ensemble des termes utilisés pour l'analyse sont définis dans le cours "Analyse et Modélisation de Systèmes d'Information" [Heymans 04].

# Section 2. Description du domaine d'application

La description du domaine correspond à l'étude de l'existant, ce que l'analyse doit considérer comme donné.

## 2.1. Description du domaine

### L'ADMINISTRATEUR

D1. L'administrateur est chargé d'assurer le bon fonctionnement de l'application BNM (l'application est décrite en page 27).
D2. L'administrateur a les droits et les connaissances qui lui donnent un accès sans restriction au Portail.
D3. L'administrateur a également les droits et les connaissances pour assurer l'administration système de l'application BNM.

### L'UTILISATEUR

D4. Un utilisateur est un opérateur qui gère la topologie et la configuration du réseau.
D5. Tout utilisateur a un accès limité au Portail: la portlet de logging et celle de monitoring.

### LES REQUETES

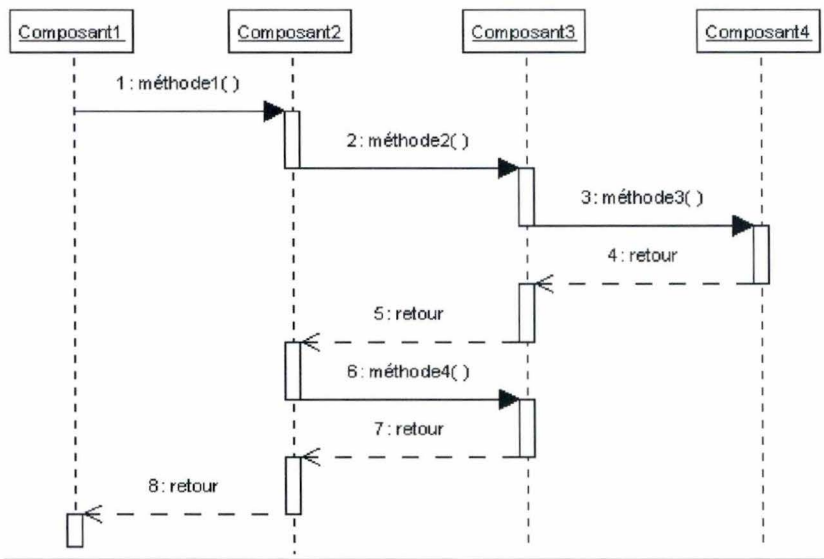



**Figure 25. Requête: exemple d'une châine de requêtes.**

D6. Le terme requête qualifie l'appel d'une méthode d'un composant de l'application BNM, de son émission jusqu'au retour de son résultat. Le diagramme de séquence de la Figure 25 contient 4 appels de méthodes. Ils peuvent être considérés comme autant de requêtes.
Seules les requêtes démarrant un traitement significatif ou susceptibles de donner des informations utiles pour ce projet sont prises en compte.
D7. L'appel à une requête de l'application BNM peut être émis à partir d'un composant étranger à l'application. L'appel de la méthode de la requête constitue donc un point d'entrée à l'application. Seuls certains composants sont susceptibles d'être appelés par un composant étranger (Portail, interface OSS/J).

D8. La requête est décomposable en une chaîne de requêtes à travers les composants de BNM. Si l'on considère tous les appels de méthodes du diagramme de séquence de la Figure 25 comme étant des requêtes, alors il s'agit d'une chaîne de requêtes composée de 4 requêtes.
D9. Toute requête générée par l'application BNM est identifiable par l'identifiant de la thread qui l'exécute.
D10. La méthode appelée caractérisant la requête peut contenir des arguments qui témoignent de l'état d'avancement de la requête. L'objet sur lequel l'appel de méthode est effectué peut aussi contenir des variables utilisables en ce sens.

## L'APPLICATION BNM

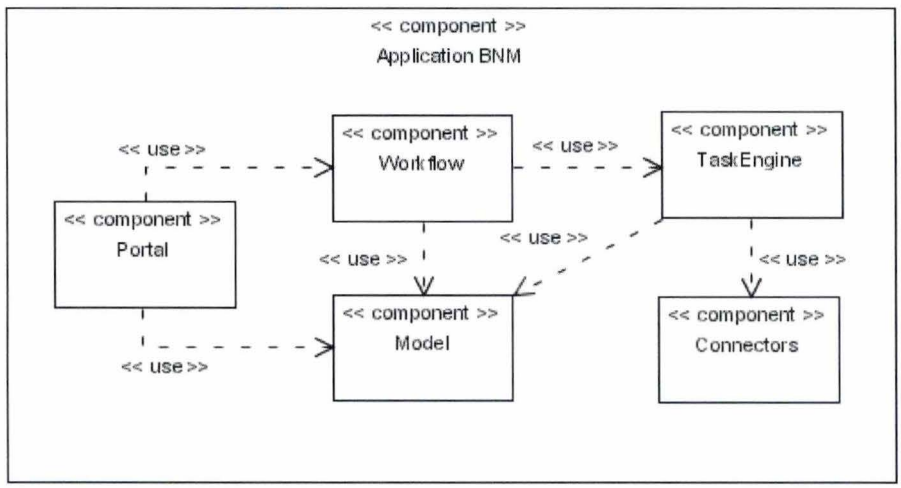



**Figure 26. Application BNM: diagramme des composants.**

### *Composants actuels*

L'application BNM est divisée en 5 composants principaux:

- **Le Portail**

D11. Le Portail gère l'accès à toutes les interfaces utilisateurs de BNM. Il s'agit d'interfaces web.
D12. Une fois authentifié, l'utilisateur a un accès limité aux portlets selon ses droits d'accès.
D13. Le fonctionnement du portail est régi par l'architecture MVC de Struts [Struts 05].
D14. Toute requête faite par un utilisateur via le Portail a la forme d'une requête http.
D15. Toute requête http est prise en charge par le serveur web intégré dans le serveur d'applications, puis transférée à la partie Contrôle de l'architecture MVC.
D16. Le Portail détient les informations sur l'émetteur de la requête émise à BNM.

- **Le Workflow**

D17. Il est construit sur base du moteur de workflow JBPM.
D18. Il fournit des services complexes sous forme de processus.
D19. Un processus du Workflow est défini par une séquence d'actions.
D20. Chaque action du processus est liée à une tâche définie dans le TaskEngine.
D21. Le Workflow est configurable. Il est possible de modifier ou d'ajouter un processus pour un type de requête déterminé.

- **Le Taskengine**

D22. C'est le composant qui gère les tâches de l'application BNM. Une tâche peut être un script ou du code Java.
D23. Chaque tâche définit un traitement spécifique utilisant une ou plusieurs ressources particulières (un Connecteur, le Modèle, ...).
D24. Une tâche peut être utilisée par une action du Workflow ou directement par un autre composant.

- **Le Modèle**

D25. Le Modèle est la base de connaissance de la topologie du réseau du client.
D26. Il comprend la configuration présente et à venir des équipements du réseau.
D27. Les informations sont consultables par les composants de BNM.
D28. Les informations sont mises à jours par les composants de BNM.
D29. Le modèle a une partie de logique métier propre à la gestion de la topologie et de la configuration du réseau.
D30. Le modèle accède à la base de données Oracle via la librairie Hibernate qui permet le mapping objet/relationnel entre les objets du modèle.

- **Les Connecteurs**

D31. Les connecteurs sont le moyen de connexion aux différents éléments du réseau.
D32. Les connecteurs sont aussi le moyen de connexion à d'éventuelles applications extérieures.

### *Composants futurs*

Il faut tenir compte que d'autres composants viendront compléter l'application. Certains ont déjà été identifiés:

- **Le Scheduler**

D33. Le Scheduler servira à différer la requête d'une tâche via le Taskengine ou d'un processus via le Workflow.
D34. Il pourra également planifier des requêtes à répétition.

- **L'interface OSS/J**

D35. L'interface OSS/J est un point d'entrée alternatif au Portail de BNM.
D36. L'interface permet à des composants externes d'émettre des requêtes à l'application BNM.

- **Le module Discovery**

D37. Le module Discovery découvre automatiquement les équipements composant le réseau.
D38. Le modèle Discovery en déduit la topologie du réseau.
D39. Le modèle Discovery renvoie la topologie au Modèle.

## 2.2. Exemple illustrant les éléments du domaine

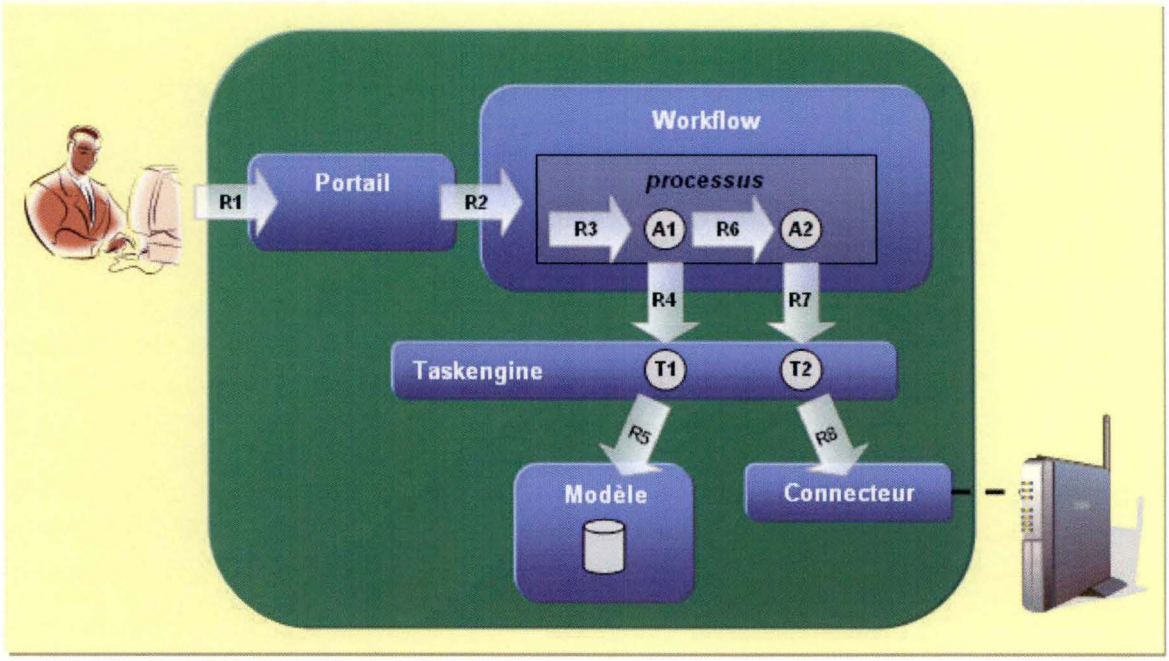



**Figure 27. Exemple de requête d'un utilisateur à l'application BNM.**

Ce petit scénario a pour seul but d'aider le lecteur à mieux appréhender les différents éléments qui composent le domaine d'application.

Un utilisateur a besoin d'effectuer une modification de la configuration d'un équipement du réseau. Il se connecte à la portlet de l'application BNM qui fournit ce type de service.
L'utilisateur remplit les informations demandées par la portlet, et soumet sa demande à l'application BNM.

Le Portail reçoit la demande de l'utilisateur et la traite. Ce traitement constitue le point de départ de la requête R1 (cf. définition de requête D6). Le traitement de la requête R1 requiert un processus du Workflow. Elle y fait donc appel (requête R2).

Supposons que le processus demandé soit composé de 2 actions A1 et A2.

La première action du processus consisterait à valider les informations entrées par l'utilisateur (vérifier l'existence de l'équipement, si la configuration de l'équipement peut être modifiée, ...).
A1 est exécutée (requête R3) et appelle la tâche qui lui est associée (cf. D20), c'est la requête R4. La validation s'effectue via le modèle pour accéder à la base de données (requête R5).
La première action A1 du processus a apporté satisfaction; la deuxième action du processus (A2) peut maintenant démarrer (R6). Celle-ci a pour but de transférer la nouvelle configuration au Connecteur via les requêtes R7 et R8.

R8 clôture son activité en retournant le résultat à la requête l'ayant initié, et ainsi de suite jusqu'à la requête de départ qui peut terminer son exécution et retourner son résultat.

# Section 3. Objectifs

Les objectifs sont aussi connus sous la dénomination de buts. Ils servent à exprimer de manière générale quelles sont les attentes du système. Celles-ci seront précisées au fil de l'analyse.

## 3.1. Objectifs fonctionnels

Obj1. **Logging**: Disposer du traçage des requêtes.
Obj2. **Monitoring**: Disposer d'un service de monitoring des requêtes.
Obj3. **Benchmarking**: Disposer d'un service de benchmarking se basant sur les données récoltées au monitoring.
Obj4. **Évaluation AOP**: Évaluer les outils AOP existant en Java.
- Obj4.1. **Implémentation AOP**: Implémenter la solution avec un outil AOP.
  - Obj4.1.1. **Tissage dynamique**: la liaison aspects – application est effectuée durant l'exécution.
  - Obj4.1.2. **Tissage statique**: à la compilation du code.

Remarque: L'Obj4 est une sorte de meta-objectif car c'est un projet qui a pour but de faire de la veille technologique.

## 3.2. Objectifs non fonctionnels

Obj5. **Performance**: Le système doit rester performant.
Obj6. **Code isolé**: Impacter au minimum le code de la logique métier du domaine.
Obj7. **Flexibilité**: L'application BNM est créée sur base d'une politique de COTS. Le nombre de composants formant l'application est donc variable. De plus, selon le client, de nouveaux Connecteurs ou même de nouveaux types de composants sont susceptibles d'être développés. Le système doit donc être facilement adaptable aux diverses évolutions et configurations.
Obj8. **Downtime minimal**: L'application doit être arrêtée le moins longtemps possible.
- Obj8.1. **Déploiement** : Éviter de redéployer l'application BNM entière en cas de mise à jour.
- Obj8.2. **Maintenance**: Éviter d'impacter l'application BNM en cas de problème.

Obj9. **Évolutivité**: Facilités de portage requises vers java1.5

## 3.3. Sélection de l'outil AOP par rapport aux objectifs

Le paradigme est réputé satisfaire aux objectifs fonctionnels du projet.
De plus, il facilite les modifications apportées après la phase de développement du système pour améliorer, ou même personnaliser la collecte des données générées par les requêtes dans l'application BNM. De plus, si le système développé n'apporte pas les résultats escomptés ou devient obsolète, sa suppression n'en sera que plus aisée.

L'évaluation des outils étant effectuée, il reste à déterminer lequel doit être utilisé. Il y a d'une part AspectJ dont le fonctionnement est basé sur le tissage statique et JBoss-AOP qui fonctionne sur base du tissage dynamique.

La Figure 28 présente le comparatif des 2 types de tissage en fonction des objectifs non fonctionnels à respecter. Il en ressort que le choix doit se faire par rapport aux performances et la disponibilité de l'application.

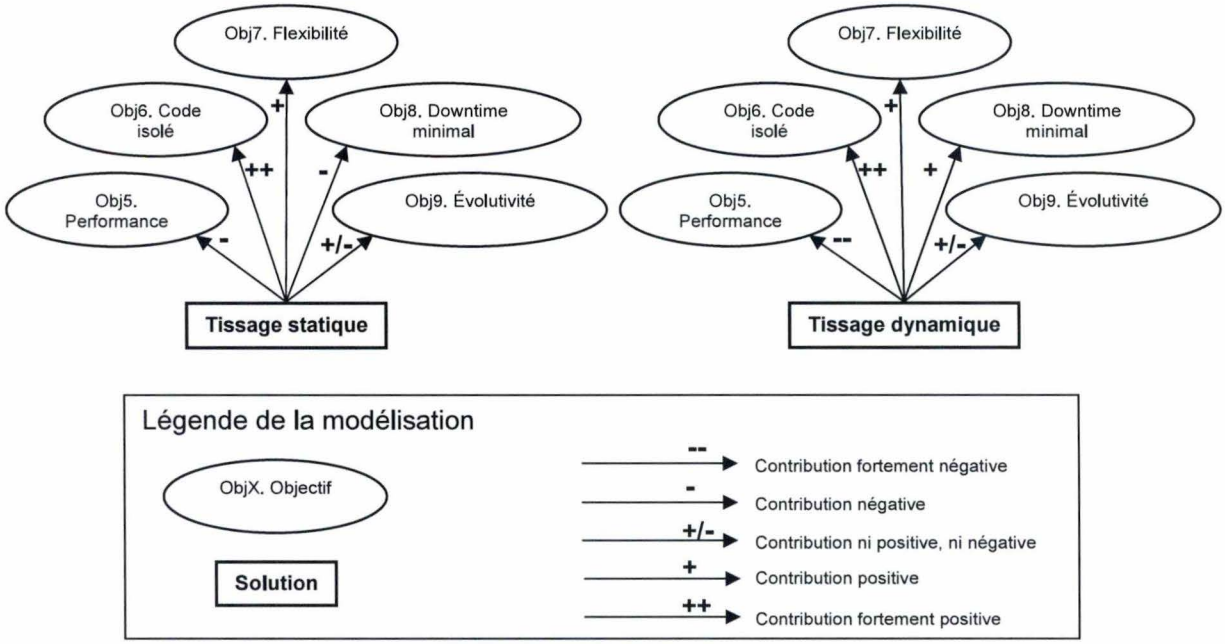



**Figure 28. Évaluation des solutions en fonction des objectifs**

Le tissage dynamique est préféré au tissage statique. Malgré les pertes de performance engendrées par cette solution, l'accent est mis sur l'importance de la disponibilité de BNM. L'application doit tourner en permanence avec le moins de perturbations possibles.

JBoss-AOP est l'outil sélectionné en ce sens. Il fait partie intégrante du serveur d'applications, il est à la base de l'architecture de la prochaine version des EJB (V.3) de JBoss, ce qui constitue une bonne garantie de suivi, d'évolution et malgré tout de performances suffisantes. JBoss-AOP semble donc prêt à atteindre l'ensemble des objectifs non fonctionnels identifiés (Obj5, Obj6, Obj7, Obj8 & Obj9).

# Section 4. Exigences

Les exigences sont le résultat de l'affinement des objectifs, essentiellement selon le point de vue de l'utilisateur.

## 4.1. Exigences fonctionnelles

Ex1. Le logging et le monitoring informent du déroulement de chaque requête, de son émission à son résultat final.
Ex2. L'utilisateur visualise les informations retournées par le logging, monitoring et benchmarking aux travers de portlets.
Ex3. L'utilisateur peut visualiser la liste résultant de la collecte du logging sous différents types de présentation.
Ex4. L'utilisateur peut filtrer les données du logging à afficher.
Ex5. L'utilisateur peut sauvegarder localement sur son PC une séquence du monitoring.
Ex6. L'utilisateur peut effectuer le benchmarking sur base des sauvegardes du monitoring.
Ex7. L'administrateur peut activer ou désactiver le service de logging.
Ex8. L'administrateur peut activer ou désactiver le service de monitoring.

***Exigences envisagée pour une prochaine itération du processus de développement,:***

Ex9. L'utilisateur peut disposer d'un rafraîchissement continu de l'état des requêtes tracées pour le logging.

Ex10. L'utilisateur peut disposer d'un rafraîchissement des données du monitoring à mesure que les requêtes utilisateurs sont émises.

Ex11. L'administrateur peut limiter la quantité de données collectées pour alléger la charge du système.

## 4.2. Exigences non-fonctionnelles

Ex12. Pour les résultats donnés par la portlet de logging, l'utilisateur différencie aisément les requêtes ayant abouti de celles qui ont échoué.

Ex13. Pour les résultats donnés par la portlet de logging, l''utilisateur différencie aisément les requêtes ayant abouti de celles qui pourraient être bloquées.

## 4.3. Matrice de traçabilité des exigences

Le tableau ci-dessous valide les exigences en vérifiant leurs relations avec les objectifs, les caractéristiques du domaine d'application et les spécifications (cf. Spécifications en page 43).
Une exigence doit provenir d'un (des) objectif(s), reposer sur des hypothèses du domaine et doit être solutionnée par une (des) spécification(s).

| Exigence | Objectifs | Domaine | Spécifications |
|---|---|---|---|
| Ex1 | Obj1, Obj2 et Obj3 | D6 à D10, D11 à D39 | Spec1, Spec2, Spec3 et Spec5 |
| Ex2 | Obj1, Obj2 et Obj3 | D4, D5, D11 à D16 | Spec5 |
| Ex3 | Obj1 | D4, D5, D11 à D16 | Spec7 |
| Ex4 | Obj1 | D4, D5, D11 à D16 | Spec6 |
| Ex5 | Obj2 | D4, D5, D11 à D16 | Spec10 |
| Ex6 | Obj2 et Obj3 | D4, D5, D11 à D16 | Spec11 |
| Ex7 | Obj1 | D1 à D3 | Spec12 |
| Ex8 | Obj2 | D1 à D3 | Spec12 |
| Ex9 | Obj1 | D4 à D39 | Spec4 |
| Ex10 | Obj2 | D4 à D39 | Spec4 |
| Ex11 | Obj5 | D1 à D3, D6 à D39 | Spec13, Spec14 |
| Ex12 | Obj1 | D4, D5, D11 à D16 | Spec8 |
| Ex13 | Obj1 | D4, D5, D11 à D16 | Spec8 |

## 4.4. Déclaration des use cases

Les use cases illustrent les exigences en décrivant des interactions-types entre le domaine et le système. Une abstraction totale est conservée quant à l'implémentation du système à développer. C'est la section "Spécifications" qui a pour rôle décrire les fonctionnalités à implémenter au niveau du système.

Les uses cases présentent des cas types, compréhensibles. Ils ne sont pas une présentation exhaustive de tous les cas d'utilisations possibles.

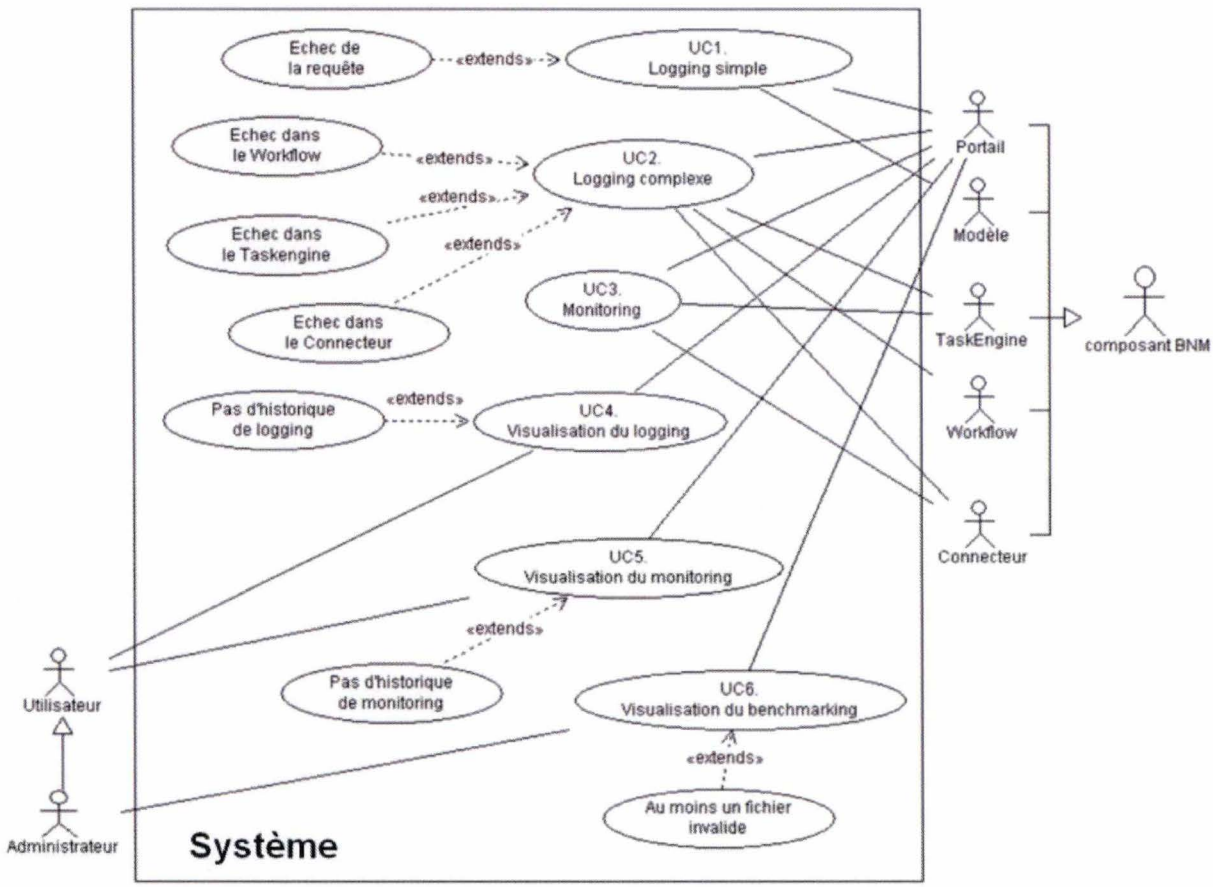



**Figure 29. Exigences: déclaration des use cases.**

Pour rappel:

- Le système est le composant de logging, monitoring et benchmarking à développer.
- BNM et l'application BNM se rapportent au projet BNM, l'application avec laquelle le système est tissé.

## 4.5. Descriptions des use cases

### USE CASE 1 : LOGGING SIMPLE

Description:
Obtention du détail du cheminement d'une requête de son émission jusqu'à la réception de son résultat. Ce cas d'utilisation illustre le cas d'un type de requête nécessitant un accès au Modèle.

Pré-conditions:
- BNM est démarré et fonctionne correctement.
- Le système fonctionne correctement.
- L'utilisateur est identifié par le Portail.
- L'utilisateur émet une requête pour obtenir des informations sur la topologie du réseau. Ces informations sont détenues dans le Modèle.

Post-conditions:
- Les informations de logging de la requête sont générées.
- L'utilisateur a reçu le résultat de la requête.

| Portail | Modèle | Système |
|---|---|---|
| 1. La requête de l'utilisateur est reçue. | | |
| | | 2. Le système génère le log de la requête. |
| | | 3. Le système collecte les informations sur l'utilisateur et sa requête (D11). |
| 4. Le portail transmet la requête au Modèle. | | |
| | | 5. Le système enregistre la requête faite au Modèle. |
| | 6. Le Modèle traite la demande et retourne le résultat au portail. | |
| 7. Le Portail renvoie le résultat à l'utilisateur. | | |
| | | 8. Le système clôture le log de la requête. |

***Cas alternatif: échec de la requête***

Description:
La requête retourne une erreur car une Exception est émise par le Modèle. Ce peut être dû à une indisponibilité de la base de données, un problème d'integrité des données, etc.

Pré-conditions:
- Idem cas normal.

Post-conditions:
- Les informations de logging de la requête sont générées.
- Le traitement de la requête est interrompu. L'utilisateur n'a pas reçu le résultat escompté.

| Portail | Modèle | Système |
|---|---|---|
| (Idem UC1 jusqu'au point 5.) | | |
| | 6. Le modèle traite la demande mais retourne une erreur. | |
| | | 7. Le système enregistre l'erreur du modèle lié à la requête. |
| 8. Le portail renvoie un message d'erreur à l'utilisateur. | | |
| | | 9. Le système clôture le log de la requête. |

## USE CASE 2: LOGGING COMPLEXE

Description:
Obtention du détail du cheminement d'une requête de son émission jusqu'à la réception de son résultat. Ce cas d'utilisation illustre le cas d'un type de requête plus complexe nécessitant un accès aux Portail, au Workflow, au Taskengine et au Connecteur.

Pré-conditions:
- BNM est démarré et fonctionne correctement.
- Le système fonctionne correctement.
- Un utilisateur est identifié par le portail.
- Une requête de configuration d'un équipement est émise par l'utilisateur. Cette requête nécessite un traitement du Workflow composé de 2 actions correspondant à 2 tâches du Taskengine.

Post-conditions:
- Les informations de logging de la requête sont générées.
- L'utilisateur a reçu le résultat de la requête.

| Portail | Workflow | Taskengine | Connecteur | Système |
|---|---|---|---|---|
| 1. La requête de l'utilisateur est reçue. | | | | |
| | | | | 2. Le système génère le log de la requête. |
| | | | | 3. Le système collecte les informations sur l'utilisateur et sa requête (D11). |
| 4. Le portail transmet la requête au Workflow. | | | | |
| | | | | 5. Le système enregistre la requête faite au Workflow. |
| | 6. Le Workflow lance la première action du processus. | | | |
| | | | | 7. Le système enregistre quelle tâche est appelée par le Workflow. |

| Portail | Workflow | Taskengine | Connecteur | Système |
|---|---|---|---|---|
| | | 8. Le Taskengine exécute la tâche. Elle nécessite une première connexion. | | |
| | | | | 9. Le système enregistre les informations de connexion utilisées par le Connecteur. |
| | | | 10. Le Connecteur transmet les informations définies par la tâche. | |
| | 11. Le Workflow lance la deuxième action du processus. | | | |
| | | | | 12. Le système enregistre quelle tâche est appelée par le Workflow. |
| | | 13. Le Taskengine exécute la tâche. Elle nécessite une deuxième connexion. | | |
| | | | | 14. Le système enregistre les informations de connexion utilisées par le Connecteur. |
| | | | 15. Le Connecteur transmet les informations définies par la tâche. | |
| | 16. Le Workflow retourne le résultat. | | | |
| 17. Le portail renvoie le résultat à l'utilisateur. | | | | |
| | | | | 18. Le système clôture le log de la requête. |

### *Cas alternatif 1: échec dans le Workflow*

Description:
La requête retourne une erreur car une Exception est émise durant le processus du Workflow (le choix de faire survenir l'Exception sur la première action est arbitraire).

Pré-conditions:
- Idem cas normal.

Post-conditions:
- Les informations de logging de la requête sont sauvegardées.
- Le traitement de la requête est interrompu. L'utilisateur n'a pas reçu le résultat escompté.

| Portail | Workflow | Taskengine | Connecteur | Système |
|---|---|---|---|---|
| (Idem UC2 jusqu'au point 5.) | | | | |
| | 6. Le Workflow lance la première action du processus, et retourne une erreur. | | | 7. Le système enregistre l'erreur du Workflow lié à la requête. |
| 8. Le Portail renvoie un message d'erreur à l'utilisateur. | | | | 9. Le système clôture le log de la requête. |

### *Cas alternatif 2: échec dans le Taskengine*

Description:
La requête retourne une erreur car une Exception est émise durant l'exécution d'une tâche du Taskengine (le choix de faire survenir l'Exception sur la première tâche est arbitraire).

Pré-conditions:
- Idem cas normal.

Post-conditions:
- Les informations de logging de la requête sont générées.
- Le traitement de la requête est interrompu. L'utilisateur n'a pas reçu le résultat escompté.

| Portail | Workflow | Taskengine | Connecteur | Système |
|---|---|---|---|---|
| (Idem UC2 jusqu'au point 7.) | | | | |
| | | 8. Le Taskengine rencontre une erreur et la retourne. | | 9. Le système enregistre l'erreur du Taskengine lié à la requête. |
| | 10. Le Workflow retourne l'erreur du Taskengine. | | | |

| | | | | |
|---|---|---|---|---|
| 11. Le Portail renvoie un message d'erreur à l'utilisateur. | | | | 12. Le système clôture le log de la requête. |

### *Cas alternatif 3: échec dans le Connecteur*

Nom:
Logging d'un échec dans le Connecteur.

Description:
La requête retourne une erreur car une Exception est émise durant l'exécution du Connecteur (le choix de faire survenir l'Exception sur le premier Connecteur est arbitraire)..

Pré-conditions:

- Idem cas normal.

Post-conditions:

- Les informations de logging de la requête sont générées.
- L'utilisateur n'a pas reçu le résultat escompté.

| Portail | Workflow | Taskengine | Connecteur | Système |
|---|---|---|---|---|
| (Idem UC2 jusqu'au point 9.) | | | | |
| | | | 10. Le Connecteur rencontre une erreur et la retourne. | 11. Le système enregistre l'erreur du Connecteur lié à la requête. |
| | 13. Le Workflow retourne l'erreur du Connecteur. | 12. Le Taskengine retourne l'erreur du Connecteur. | | |
| 14. Le Portail renvoie un message d'erreur à l'utilisateur. | | | | 15. Le système clôture le log de la requête. |

## USE CASE 3: MONITORING

Description:
Pour illustrer ce use case, le monitoring est effectué sur une requête nécessitant le traitement d'une tâche du Taskengine.

Pré-conditions:
- BNM est démarré et fonctionne correctement.
- Le système fonctionne correctement.
- L'utilisateur est identifié par le Portail.
- Une requête d'informations nécessitant l'usage du Taskengine et d'un Connecteur est émise par l'utilisateur.

Post-conditions:
- Les informations de monitoring de la requête sont générées.
- L'utilisateur a reçu le résultat de la requête.

| Portail | Taskengine | Connecteur | Système |
|---|---|---|---|
| 1. La requête de l'utilisateur est reçue. | | | |
| | | | 2. Le système démarre le monitoring de la requête faite au Portail. |
| 3. Le portail transmet la requête au Taskengine. | | | |
| | | | 2. Le système démarre le monitoring de la requête faite au Taskengine. |
| | 4. Le Taskengine exécute la tâche demandée nécessitant une connexion. | | |
| | | | 5. Le système créée une session de monitoring sur la requête faite au Connecteur. |
| | | 6. Le Connecteur transmet les informations définies par la tâche. | |
| | | | 7. Le système clôture la session de monitoring sur le Connecteur. |
| | | | 8. Le système clôture la session de monitoring sur le Taskengine. |
| 9. Le Portail renvoie le résultat à l'utilisateur. | | | |
| | | | 10. Le système clôture la session de monitoring sur le Portail. |

Remarque: Quel que soit le résultat de la requête (réussite, échec ou exception), le traitement du monitoring reste le même car ce sont les temps d'exécution qui importent.

## USE CASE 4: VISUALISATION DU LOGGING

Description:
Un utilisateur connecté au Portail lance la portlet de logging, visualise les logs, affine les résultats, effectue une sauvegarde de la séquence choisie et en charge une autre.

Pré-conditions:
- BNM est démarré et fonctionne correctement.
- Le système fonctionne correctement.
- L'utilisateur est identifié par le Portail.

Post-conditions:
- L'utilisateur a visualisé l'historique du logging souhaité.
- L'utilisateur a un fichier texte sur son PC contenant l'historique du logging souhaité.

| Utilisateur | Système<br>(interface : PC de l'utilisateur, portlet de logging) |
|---|---|
| 1. L'utilisateur lance l'interface de visualisation du logging. | |
| | 2. Le système affiche l'interface utilisateur de logging. |
| 3. L'utilisateur choisit de visualiser l'historique du logging de la veille. | |
| | 4. Le système renvoie l'historique du logging de la veille trié par requête, par date pour la période demandée. |
| 5. L'utilisateur demande de sauvegarder cet historique du logging. | |
| | 6. Le système sauve l'historique du logging en local sur le PC de l'utilisateur. |
| 7. L'utilisateur demande de charger une autre sauvegarde. | |
| | 8. Le système renvoie l'historique du logging du fichier spécifié trié par requête, par date. |

### *Cas alternatif: Pas d'historique de logging*

Description:
L'utilisateur demande l'historique du logging sur une période où les logs ne sont pas disponibles.

Pré-conditions:
- BNM est démarré et fonctionne correctement.
- Le système fonctionne correctement.
- L'utilisateur est identifié par le Portail.

Post-conditions:
- L'utilisateur n'a pas pu visualiser l'historique du logging souhaité.

| Utilisateur | Système<br>(interface : PC de l'utilisateur, portlet de logging) |
|---|---|
| (Idem UC4 jusqu'au point 3.) | |
| | 4. Le système renvoie un message signalant l'indisponibilité de la période désirée. |

## USE CASE 5: VISUALISATION DU MONITORING

Description:
Un utilisateur connecté au Portail lance la portlet de monitoring, visualise le monitoring, effectue une sauvegarde des compteurs du monitoring et en charge une autre.

Pré-conditions:
- BNM est démarré et fonctionne correctement.
- Le système fonctionne correctement.
- L'utilisateur est identifié par le Portail.

Post-conditions:
- L'utilisateur a visualisé les résultats du monitoring souhaité.
- L'utilisateur a un fichier texte sur son PC contenant le résultat du monitoring souhaité.

| Utilisateur | Système<br>(interface : PC de l'utilisateur, portlet de logging) |
|---|---|
| 1. L'utilisateur lance l'interface de visualisation du monitoring. | |
| | 2. Le système affiche l'interface utilisateur du monitoring. |
| 3. L'utilisateur choisit le résultat du monitoring de la veille. | |
| | 4. Le système calcule les compteurs et affiche le résultat du monitoring pour la période demandée. |
| 5. L'utilisateur demande de sauvegarder le résultat du monitoring. | |
| | 6. Le système sauve le résultat du monitoring en local sur le PC de l'utilisateur. |
| 7. L'utilisateur demande de charger une autre sauvegarde. | |
| | 8. Le système renvoie le résultat du monitoring du fichier spécifié. |

### *Cas alternatif: Pas d'historique de monitoring*

Description:
L'utilisateur demande le résultat du monitoring sur une période où il n'est pas disponible.

Pré-conditions:
- BNM est démarré et fonctionne correctement.
- Le système fonctionne correctement.
- L'utilisateur est identifié par le Portail.

Post-conditions:
- L'utilisateur n'a pas pu visualiser le résultat du monitoring souhaité.

| Utilisateur | Système<br>(interface : PC de l'utilisateur, portlet de logging) |
|---|---|
| (Idem UC5 jusqu'au point 3.) | |
| | 4. Le système renvoie un message signalant l'indisponibilité de la période désirée. |

## USE CASE 6: VISUALISATION DU BENCHMARKING

Description:
Un utilisateur connecté au Portail lance la portlet de benchmarking et effectue la comparaison de deux fichiers de monitorings.

Pré-conditions:
- BNM est démarré et fonctionne correctement.
- Le système fonctionne correctement.
- L'utilisateur est identifié par le Portail.
- L'utilisateur dispose de deux fichiers issus de deux sauvegardes de monitoring.

Post-conditions:
- L'utilisateur a visualisé les résultats du benchmarking.

| Utilisateur | Système<br>(interface : PC de l'utilisateur, portlet de logging) |
|---|---|
| 1. L'utilisateur lance l'interface de visualisation du benchmarking. | |
| | 2. Le système affiche l'interface utilisateur du benchmarking. |
| 3. L'utilisateur spécifie au système les deux fichiers de monitoring devant être comparés. | |
| | 4. Le système charge les deux fichiers et les compare.<br>5. Le système affiche le résultat de la comparaison. |

### *Cas alternatif: Au moins un fichier invalide*

Description:
Un utilisateur connecté au Portail lance la portlet de benchmarking et effectue la comparaison de deux fichiers de monitorings dont au moins un des deux n'est pas le résultat d'un monitoring.

Pré-conditions:
- BNM est démarré et fonctionne correctement.
- Le système fonctionne correctement.
- L'utilisateur est identifié par le Portail.
- L'utilisateur dispose de deux fichiers mais au moins l'un d'entre eux n'est pas un fichier généré par le monitoring.

Post-conditions:
- L'utilisateur a reçu un message d'erreur.

| Utilisateur | Système<br>(interface : PC de l'utilisateur, portlet de logging) |
|---|---|
| (Idem UC6 jusqu'au point 3.) | |
| | 4. Le système charge les deux fichiers et détecte que le format d'au moins un des deux est invalide.<br>5. Le système affiche un message d'erreur désignant le ou les fichiers erronés et invitant à recommencer la procédure. |
| (Retour à l'UC6, point 3.) | |

# Section 5. Spécifications

Les spécifications traduisent les exigences des utilisateurs dans le but de guider le travail des programmeurs.

## 5.1. Les données

Spec1. Les données utilisées lors des requêtes (D10) sont interceptées par des aspects de gestion de traces (Obj4.1).
Spec2. Les données collectées (D10) sont stockées de manière durable.
Spec3. Les données stockées sont utilisées par les composants de visualisation.

***Spécification envisagée pour une prochaine itération du processus de développement:***
Spec4. Les données collectées sont transférées directement aux interfaces utilisateurs connectés.

## 5.2. La visualisation du logging

Spec5. Quand l'utilisateur en fait la requête via la portlet de logging, les données collectées souhaitées sont transférées puis affichées à l'utilisateur.
Spec6. Pour filtrer les données à afficher, l'utilisateur sélectionne les composants qui doivent être affichés à l'écran. Le système ne retourne alors que le log des requêtes de ces composants.
Spec7. La portlet de logging propose différentes alternatives pour choisir la mise en forme du résultat.
Spec8. La portlet de logging met en évidence à l'aide de couleurs les éléments les requêtes bloquées ou ayant échouées.

## 5.3. La visualisation du monitoring

Spec9. Quand l'utilisateur en fait la requête via la portlet de monitoring, les données collectées souhaitées sont transférées puis affichées à l'utilisateur.
Spec10. La portlet de monitoring permet de sauver les données en local chez l'utilisateur.

## 5.4. La visualisation du benchmarking

Spec11. L'utilisateur spécifie 2 fichiers de monitoring à comparer pour voir le résultat du benchmarking. Chaque valeur affichée est dans une couleur déterminée en fonction de la comparaison.

## 5.5. La visualisation de l'administrateur

Spec12. Retirer le système du répertoire de déploiement du serveur d'applications a pour effet de désactiver tout le système. Redéployer le composant la réactive.

***Spécifications envisagées pour une prochaine itération du processus de développement:***
Spec13. L'administrateur peut limiter la charge de données collectées en sélectionnant un niveau de priorité lié aux évènements tracés.
Spec14. L'administrateur peut limiter la charge de traitement du système en désactivant le traçage de certains composants.

# Section 6. Conception logique

Nous avons choisi assez naturellement de structurer le système en 3 composants distincts:

- Le composant Aspect pour la collecte des données de logging et de monitoring.
- Le composant Stockage pour le stockage des données.
- Le composant Visualisation qui constitue l'ensemble des GUI's sous forme de portlets dans le portail.

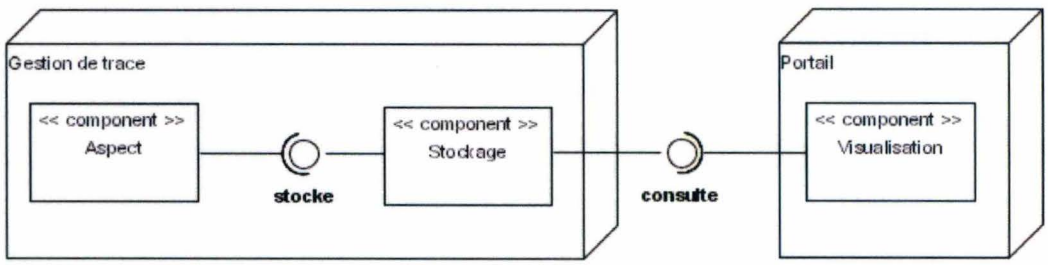



**Figure 30. Conception logique: diagramme des composants.**

Le composant Aspect transmet les informations collectées au composant Stockage tandis que le composant Visualisation les utilise et les affiche.

## 6.1. La collecte des données

### LES STATUTS D'UNE REQUETE

Conformément à l'objectif Obj4.1, les données sont collectées à partir du composant Aspect dont la charge est de fournir des informations liées aux évènements propres aux requêtes.

Chaque instance de type Request a pour fonction de conserver les informations propres à une requête. Ces informations sont collectées et mises à jour par le composant Aspect tout au long de l'exécution de la requête. Voici les états possibles d'une requête que doit gérer l'objet de type Request. Ceux-ci évoluent selon les évènements générés par le composant Aspect.

Remarque: saveAllValues n'influence pas l'état Started car cet évènement a trait à la persistance des données et non à la requête.

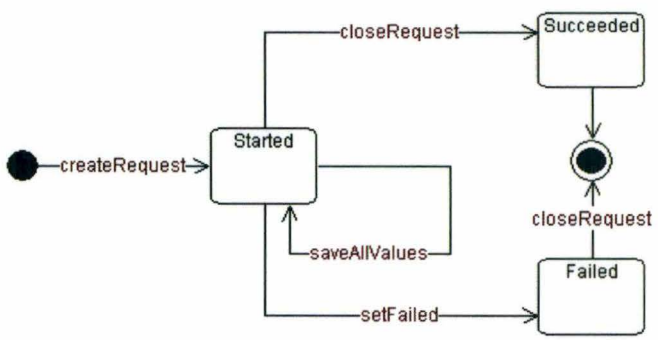



**Figure 31. Conception logique: diagramme d'état de la classe Request.**

**Started**
La création d'une requête signifie qu'une requête est lancée, son suivi a commencé.

**Failed**
Un évènement setFailed a été émis, la requête a donc échoué et sera clôturée par l'évènement closeRequest dans l'état failed.

**Succeeded**
L'évènement closeRequest est survenu sans qu'une erreur ne survienne. La requête est donc clôturée dans l'état succeeded.

## LES EVENEMENTS

Voici l'inventaire des évènements possibles et utiles que peut générer le composant Aspect. Ils sont tous illustrés dans le Use case: Traçage de deux requêtes ainsi que dans ses cas alternatifs (pages 48 à 52).

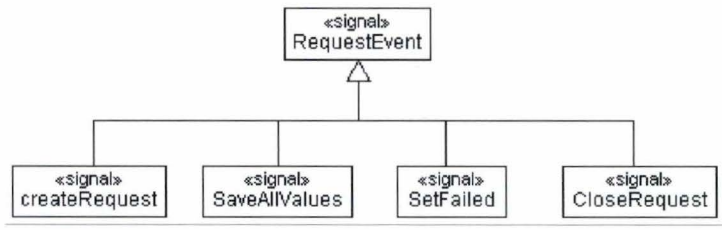



**Figure 32. Conception logique: diagramme des évènements de la classes Request.**

**RequestEvent**
C'est la généralisation de l'ensemble des évènements relatifs à toute requête exécutée dans l'application BNM.

**CreateRequest**
L'évènement est généré lorsqu'une nouvelle requête est émise. La création de la requête est sauvegardée pour qu'une trace de la requête subsiste en cas de problème.

**SaveAllValues**
L'évènement sert à sauvegarder les données collectées propres au type de la requête.

**SetFailed**
L'évènement change l'état de Request à Failed car un problème s'est produit durant l'exécution de la requête. Le message d'erreur éventuel est sauvegardé.

**CloseRequest**
L'évènement fournit le moment exact de la fin de la requête et signale au système que la requête est clôturée.

L'étude du code source de l'application[5] a révélé les pointcuts fournissant des informations significatives. Voici l'ensemble des données pouvant être collectées pour chaque pointcut.

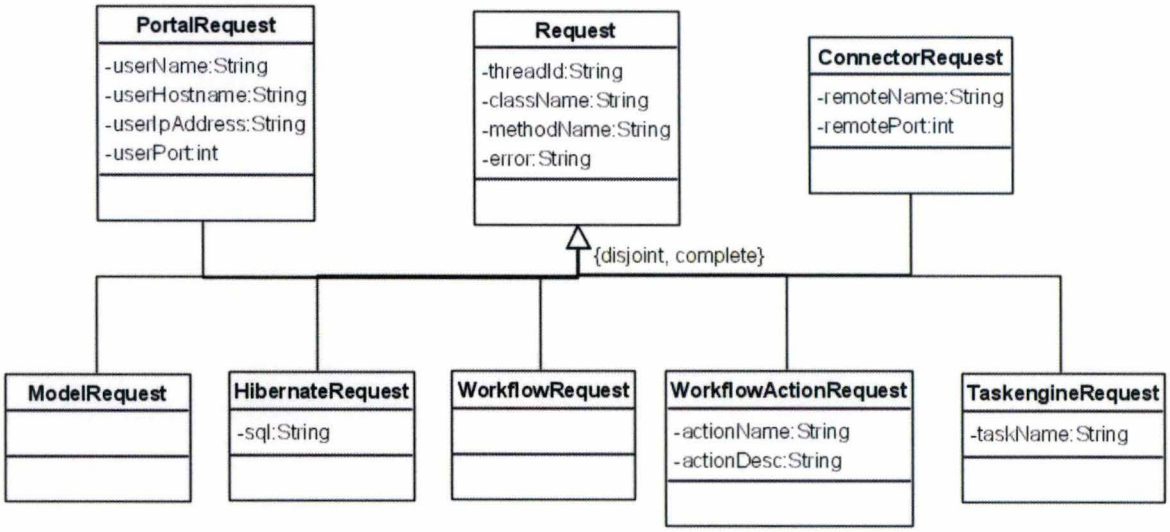



**Figure 33. Conception logique: diagramme de classes des données collectées pour le logging.**

### *Pour tout type de poincut.*

- **threadId** L'identifiant du Thread courant (currentThread).
- **className** Le nom de la classe à l'origine de la requête.
- **methodName** Le nom de la méthode à l'origine de la requête.
- **error** Tout éventuelle erreur survenant dans une requête.

### *Pour le portail*

- **userName** Le login du user.
- **userHostName** Le nom du PC de l'utilisateur.
- **userIpAddress** L'adresse IP du PC de l'utilisateur.
- **userPort** Le numéro de port de la machine du client.

### *Pour le Modèle*

(Uniquement les données communes à tout type de pointcut)

### *Pour Hibernate*

- **sql** La requête SQL faite en base de données.

### *Pour le processus du Workflow*

(Uniquement les données communes à tout type de pointcut)

### *Pour toute action du Workflow*

- **action** Le nom de chaque action exécutée.
- **description** La description de chaque action exécutée.

[5] Le code source de l'application BNM n'est pas disponible dans ce document en vertu des clauses de confidentialité de Micro Research S.A.

### *Pour toute tâche du Taskengine*

- **taskName** Le nom de la tâche à exécuter.

### *Les Connecteurs*

- **remoteName** Le nom de l'hôte auquel on se connecte.
- **remotePort** Le port de cet hôte.

## INVENTAIRE DES DONNEES COLLECTEES POUR LE MONITORING

Les données utilisées pour le monitoring ne sont pas directement collectées. Il s'agit plutôt de compteurs déduits des évènements émis par le composant Aspect.

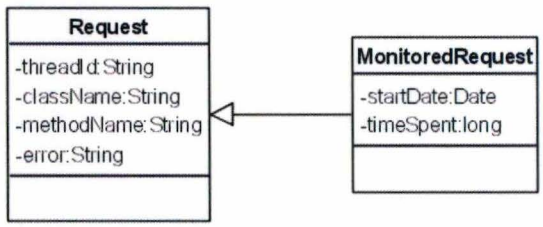



**Figure 34. Conception logique: la classe illustrant les données du monitoring.**

Pour chaque pointcut sont récoltées les données suivantes:

- **startDate** La date d'exécution de chaque requête (java.util.Date).
- **timeSpent** Le temps passé (en millisecondes) depuis l'exécution de la requête jusqu'à sa clôture.

## LA CLASSE REQUEST

Puisque les données du logging et du monitoring concernent les mêmes requêtes, celles-ci sont dorénavant rassemblées dans une classe unique Request.

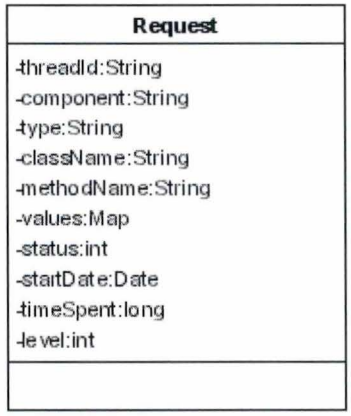



**Figure 35. Conception logique: la classe Request.**

On y retrouve:

- **threadId, className** et **methodName** Les données du logging communes à tout type de poincut (cf. page 46).
- **startDate, timeSpent** Les données du monitoring (cf. ci-dessus).

- **values** Les données du logging spécifiques à chaque poincut (cf. page 46) y sont stockées[6].
- **status** le statut de la requête (cf. page 44).

La classe est aussi enrichie par d'autres données qui serviront à classer les requêtes:
- **component** Le nom du composant qui exécute la requête. Cette donnée est déduite du pointcut utilisé.
- **type** Le type de requête émise. La définition du type est différente en fonction de la méthode à l'origine de l'exécution de la requête.

| Composant | Type |
|---|---|
| Connecteur | Type unique: envoi d'une commande. |
| Taskengine | Le nom de la tâche. |
| Hibernate | Type unique: envoi d'une requête SQL. |
| Model | Le nom de la méthode exécutée. |
| Workflow | Type unique: lancement des actions du flow. |
| WorkflowAction | Le nom de l'action exécutée. |

Un champ est aussi ajouté spécialement pour l'utilisation de chaînes de requêtes (cf. "Implémentation d'une chaîne de requêtes" en page 53):
- **level** le niveau d'appel par rapport à la chaîne de requêtes.

Maintenant que les composants, la nature de leurs interactions et les données traitées ont été identifiés, il devient possible d'illustrer cette étape de conception avec des use cases.

### USE CASE: TRAÇAGE DE DEUX REQUETES

Description:
Ce diagramme de séquence illustre l'évolution de 2 requêtes, de leur création à leur clôture.
Un utilisateur effectue une demande d'informations sur la topologie du réseau via son PC. Sa demande génère une première requête sur le Portail qui génère la seconde au niveau du Modèle.

Pré-conditions:
- BNM est démarré et fonctionne correctement.
- Le système fonctionne correctement.
- L'utilisateur est identifié par le Portail.

Post-conditions:
- Les informations de logging et de monitoring de la requête sont générées.
- L'utilisateur a reçu le résultat de la requête.

[6] La Map values rend le code source du système plus générique: tout ajout ou retrait de données spécifiques à un poincut ne nécessite qu'une seule modification du code source au niveau de l'aspect qui génère l'objet de type Request.

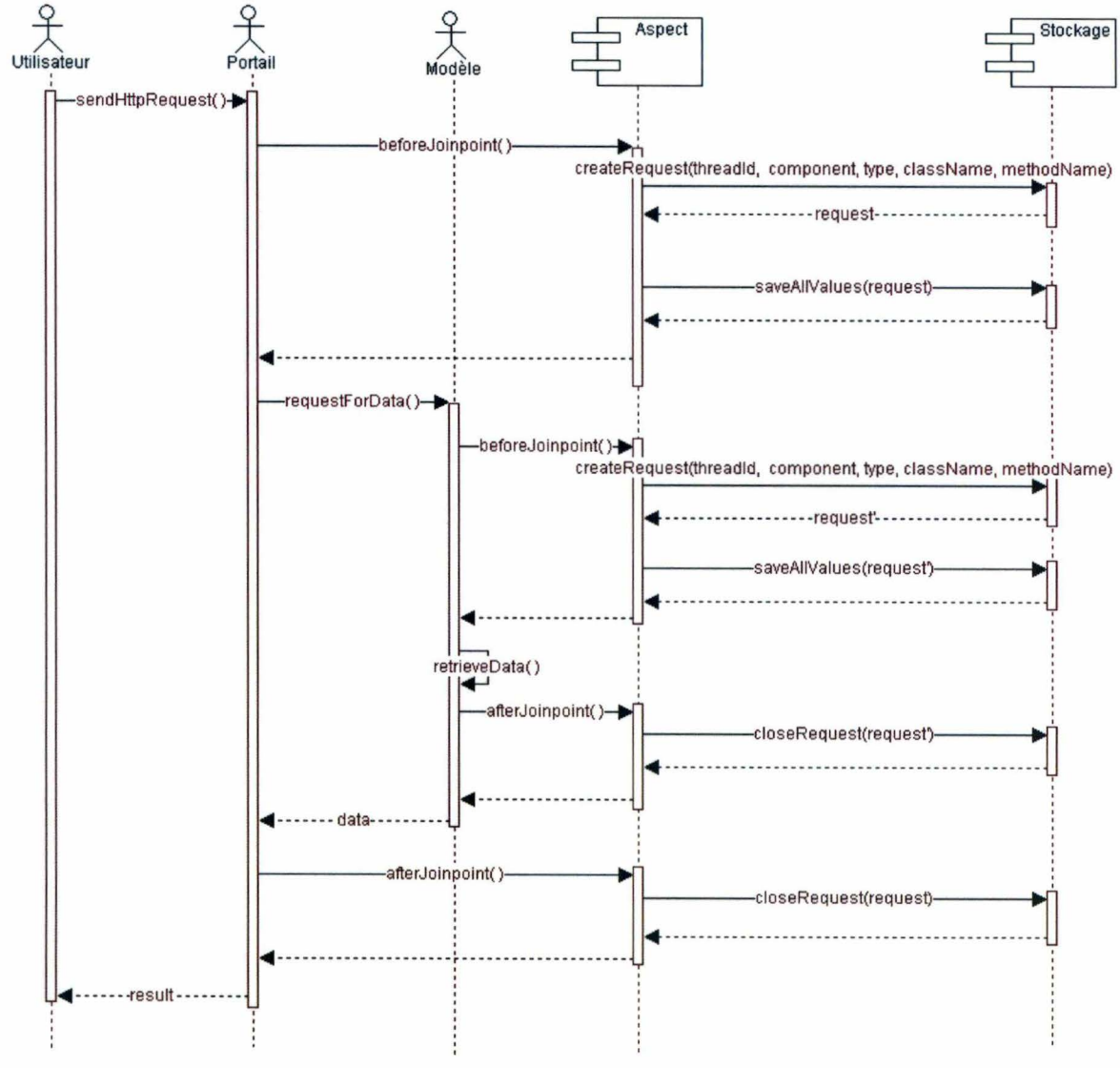



**Figure 36. Conception logique: use case du traçage de deux requêtes.**

Remarque: beforeJoinpoint et afterJoinpoint sont émis par les composants de BNM à partir du moment où ils ont été instrumentés. Ce choix de représentation a l'avantage de conserver la séquence des évènements.

**sendHttpRequest**

- Utilité: La méthode illustre l'émission d'une requête http au Portail par l'utilisateur.
- Paramètres: Aucun.
- Pré-condition: L'utilisateur est prêt à soumettre sa requête http à l'aide de son browser Internet.
- Post-condition: La requête a été traitée, et l'écran de résultat est affiché à l'utilisateur.

**beforeJoinpoint**

- Utilité: Un joinpoint est atteint. Il déclenche l'aspect avant que le code de la méthode de la logique métier pointée ne soit exécuté.
- Paramètres: Aucun.
- Pré-condition: Un appel est survenu à la méthode ciblée par le joinpoint.
- Post-condition: Le code de l'advice lié a été exécuté.

**createRequest**

- Utilité: L'évènement de création de requête (défini dans "Les évènements" en page 45).
- Paramètres: threadId, component, type, className et methodName (cf."La classe Request" en page 47 où toutes ces données sont définies).
- Pré-condition: Les données nécessaires à l'identification de la requête sont collectées par le composant Aspect.
- Post-condition: Une instance de type Request est retournée. Il contient les données passées en paramètres, son état est Started (défini dans "Les statuts d'une requête" en page 44), et sa date de démarrage. Toutes les informations relatives à cette nouvelle requête sont sauvegardées par le composant Stockage.

**saveAllValues**

- Utilité: L'évènement de sauvegarde des valeurs propres au composant de la requête (défini dans "Les évènements" en page 45).
- Paramètres: L'instance de type Request créée précédemment durant l'exécution du même advice.
- Pré-condition: La méthode createRequest a retourné l'instance de type Request, les données à sauvegarder ont été collectées.
- Post-condition: Les données collectées depuis la création de la requête ont été sauvegardées par le composant Stockage.

**requestForData**

- Utilité: La méthode à l'origine de la requête de l'utilisateur au niveau du Modèle.
- Paramètres: Aucun.
- Pré-condition: Le portail appelle cette méthode suite à la requête http d'un utilisateur. Le composant Aspect a intercepté l'appel et a exécuté un advice.
- Post-condition: La logique métier nécessaire à la requête de l'utilisateur est effectuée. Le résultat retourné.

**retrieveData**

- Utilité: La méthode illustrant la logique métier utilisée pour traiter la requête de l'utilisateur au niveau du Modèle.
- Paramètres: Aucun.
- Pré-condition: Traitement de la méthode requestForData.
- Post-condition: Le traitement relatif à la logique métier est effectué.

**afterJoinpoint**

- Utilité: La méthode pointée par le joinpoint s'est terminée. L'aspect termine l'exécution de son advice.
- Paramètres: Aucun.
- Pré-condition: Le traitement relatif à la logique métier du joinpoint est effectué.
- Post-condition: Le code de l'advice lié a été exécuté et l'objet résultant de la méthode interceptée (requestForData) est retourné.

**closeRequest**

- Utilité: L'évènement de clôture de requête (défini dans "Les évènements" en page 45).
- Paramètres: L'instance de type Request créée précédemment durant l'exécution du même advice.
- Pré-condition: Le traitement de la requête est terminé et celle-ci peut être clôturée.
- Post-condition: L'instance de type Request est passé à l'état Succeeded (défini dans "Les statuts d'une requête" en page 44) et le temps total depuis sa création est calculé. Ces informations sont aussi mises à jour dans le composant Stockage.

### *Cas alternatif 1: Échec lors de l'exécution de la requête du Portail*

Description:
Cas où la requête du Portail rencontre une erreur avant d'émettre la requête au Modèle.

Pré-conditions:

- BNM est démarré et fonctionne correctement.
- Le système fonctionne correctement.
- L'utilisateur est identifié par le Portail.

Post-conditions:

- Les informations de logging et de monitoring de la requête sont générées pour le Portail.
- L'utilisateur a reçu un message d'erreur.

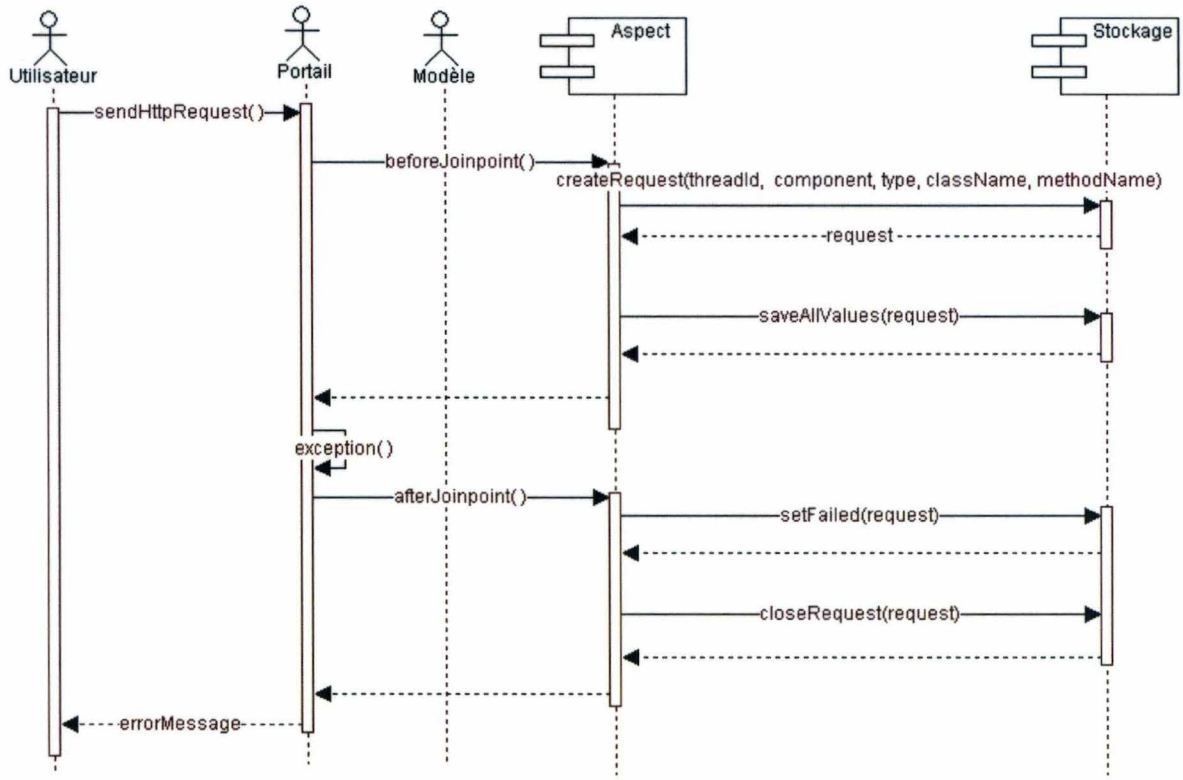



**Figure 37. Conception logique: use case du traçage de deux requêtes en échec au Portail.**

**exception**

- Utilité: La méthode illustrant la survenue d'un problème lors du traitement en cours.
- Paramètres: Aucun.
- Pré-condition: Aucune.
- Post-condition: Une erreur détectable par le composant Aspect est survenue.

**setFailed**

- Utilité: L'évènement mettant le statut de la requête en erreur (défini dans le chapitre "Les évènements" en page 45).
- Paramètres: L'instance de type Request créée précédemment durant l'exécution du même advice.
- Pré-condition: Le traitement de la requête a rencontré un problème. Le composant Aspect l'a détecté.
- Post-condition: L'instance de type Request est passée à l'état Failed (défini dans "Les statuts d'une requête" en page 44) et le temps total de son exécution est calculé. Ces informations sont aussi mises à jour dans le composant Stockage.

### *Cas alternatif 2: Échec lors de l'exécution de la requête du Modèle*

Description:
Cas où la requête du Modèle rencontre une erreur.

Pré-conditions:

- BNM est démarré et fonctionne correctement.
- Le système fonctionne correctement.
- L'utilisateur est identifié par le Portail.

Post-conditions:

- Les informations de logging et de monitoring de la requête sont générées pour le Portail.
- L'utilisateur a reçu un message d'erreur.

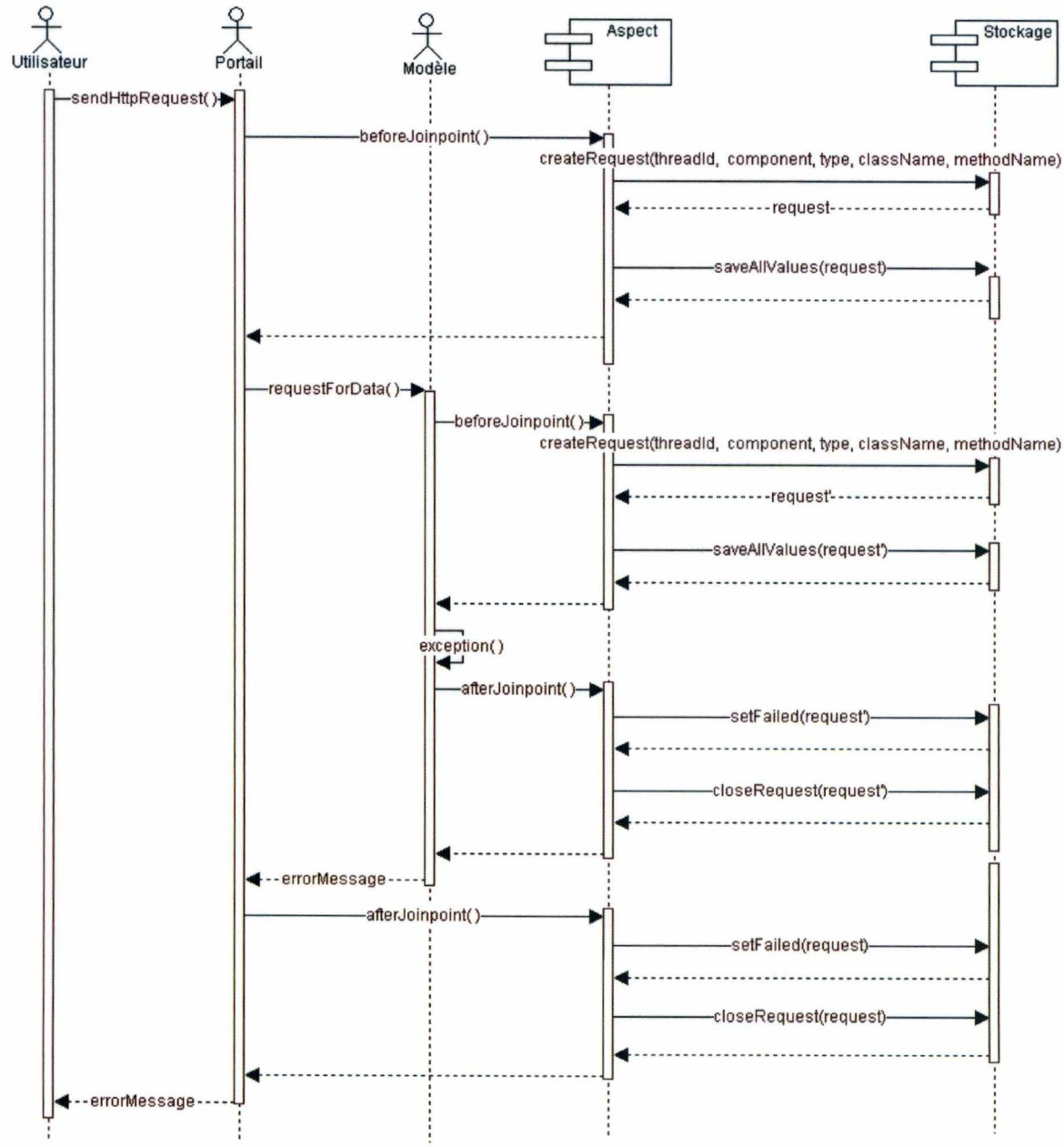



**Figure 38. Conception logique: use case du traçage de deux requêtes en échec au Modèle.**

## 6.2. La visualisation du logging

### VISUALISATION DU RESULTAT

Sachant qu'une requête peut aussi être une chaîne de requêtes (cf. D8 en page 27), il est intéressant de pouvoir visualiser l'historique des requêtes sous ces 2 formes.

Pour rappel (cf. "La classe Request" en page 47):
- chaque requête est un objet de type Request,
- chaque chaîne de requêtes est une liste d'objets de type Request.

### IMPLEMENTATION D'UNE CHAINE DE REQUETES

Pour rappel, une chaîne de requêtes est une succession d'appels de méthodes jusqu'à ce que la première méthode émise reçoive son résultat (cf. D8 en page 27).

Dans les cas simples, cette succession d'appels est linéaire, chaque requête engendre une nouvelle requête. L'implémentation d'une liste d'objet Request suffit pour représenter ce type de chaîne.

Dans les cas plus compliqués, une même requête peut engendrer plusieurs requêtes, et donc plusieurs "sous-chaînes" de requêtes distinctes. On devrait donc être dans le cas d'une arborescence d'objets de type Request.
Cependant, il a été choisi de conserver une structure de liste d'objets de type Request pour faciliter l'affichage de la chaîne dans une page HTML. Pour y arriver, un champ **level** est ajouté à la classe Request.

Regardons la valeur de level pour chaque requête de cet exemple repris de la Figure 25.

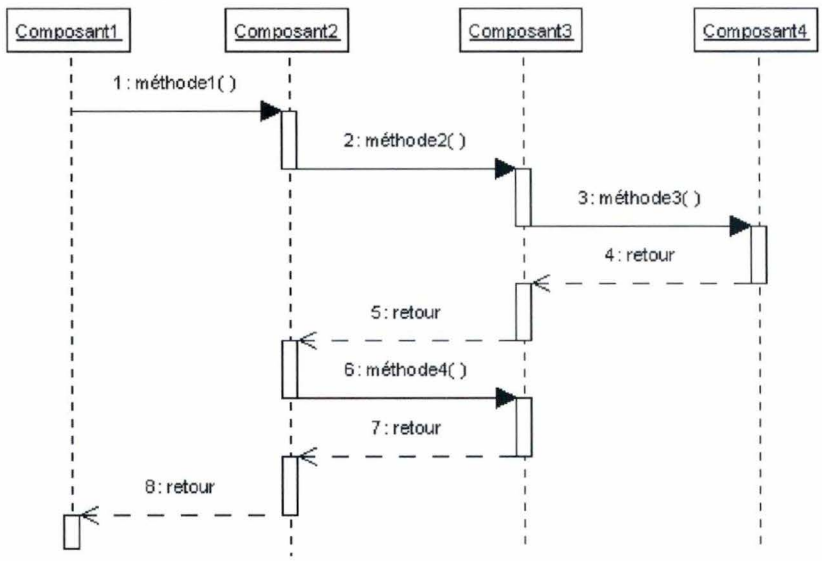



**Figure 39. Conception logique: illustration d'une chaîne de requêtes.**

L'appel à la méthode1 constitue la première requête donc le level est à 1.
Les appels aux méthode2 et méthode4 sont 2 requêtes issues de la méthode1, donc leur level est à 2.
L'appel à la méthode3 est en level 3 puisqu'il provient de méthode2 en level 2.

## LES FILTRES

Conformément à la Spec6 (en page 43), l'utilisateur doit pouvoir disposer de filtres:

- Un filtre pour désigner l'intervalle de temps dans lequel les requêtes ont généré des traces dans le composant Stockage.
- Le composant auquel appartient la requête.
- Le statut des requêtes.

Ces 3 filtres sont à utiliser pour la visualisation par requête. Par contre, la visualisation par chaîne de requêtes ne doit pas utiliser le filtre basé sur les composants car une chaîne de requêtes passe généralement par la plupart des composants disponibles.

## USE CASE: VISUALISATION DU LOGGING PAR REQUETE

Description:
Ce diagramme de séquence illustre les interactions entre le composant Visualisation et Stockage lorsqu'un utilisateur demande à visualiser l'historique du logging par requête.

Pré-conditions:

- BNM est démarré et fonctionne correctement.
- Le système fonctionne correctement.
- L'utilisateur est identifié par le Portail et a atteint la portlet de logging.

Post-conditions:

- L'utilisateur visualise l'historique des requêtes triées et filtrées selon ses besoins.

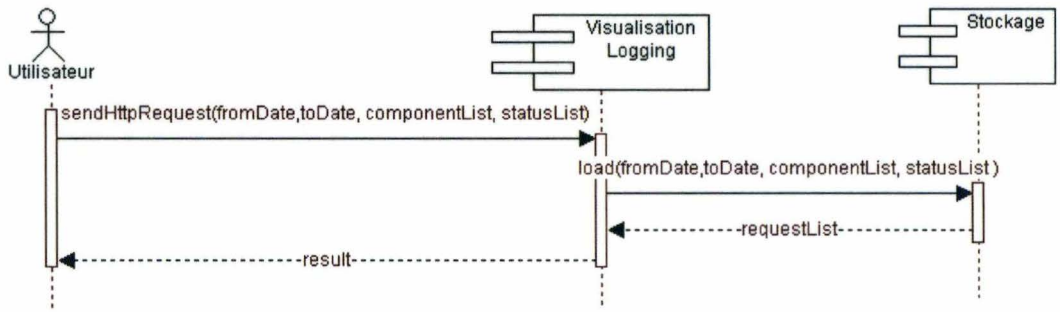



**Figure 40. Conception logique: use case de visualisation du logging par requête.**

**sendHttpRequest**

- Utilité: La méthode illustre l'émission de la requête http faite par l'utilisateur pour obtenir l'historique des requêtes selon les critères désirés.
- Paramètres: La date de départ (fromDate), la date de fin (toDate), la liste des composants à tenir en compte (componentList) et les statuts de requêtes (statusList).
- Pré-condition: L'utilisateur est prêt à soumettre sa requête http à l'aide de son browser Internet.
- Post-condition: La requête a été traitée, et l'écran de résultat est affiché à l'utilisateur.

**load**

- Utilité: Chargement de l'historique des requêtes.
- Paramètres: La date de départ (fromDate), la date de fin (toDate), la liste des composants à tenir en compte (componentList) et les statuts de requêtes (statusList).
- Pré-condition: L'utilisateur soumet sa requête http à l'aide de son browser Internet.
- Post-condition: L'historique des requêtes est retourné sous forme d'une liste chronologique d'objets de type Request.

***Cas alternatif: Les dates de début et de fin sont invalides.***

Description:
Cas où l'utilisateur spécifie une date de départ postérieure à la date de fin, ou a omis de spécifier au moins l'une des 2 dates.

Pré-conditions:
- BNM est démarré et fonctionne correctement.
- Le système fonctionne correctement.
- L'utilisateur est identifié par le Portail et a atteint la portlet de logging.

Post-conditions:
- L'utilisateur visualise un message l'invitant à entrer correctement les dates demandées.

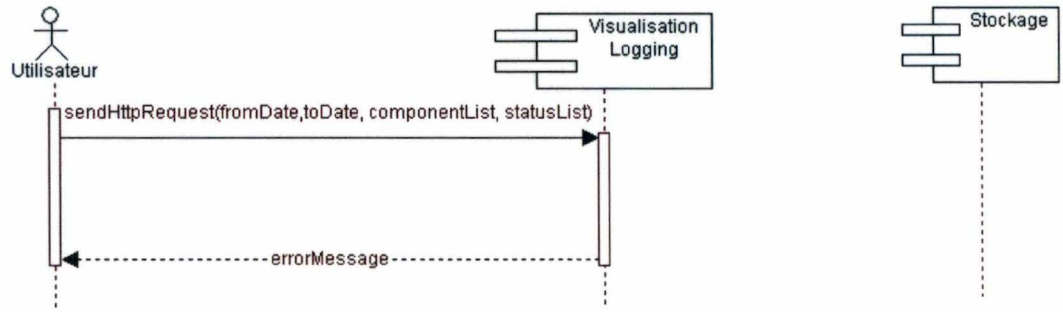



**Figure 41. Conception logique: use case de visualisation du logging par requête en échec.**

## CASE: VISUALISATION DU LOGGING PAR CHAINE DE REQUETES

Description:
Ce diagramme de séquence illustre les interactions entre le composant Visualisation et Stockage lorsqu'un utilisateur demande à visualiser l'historique du logging par chaîne de requêtes.

Pré-conditions:
- BNM est démarré et fonctionne correctement.
- Le système fonctionne correctement.
- L'utilisateur est identifié par le Portail et a atteint la portlet de logging.

Post-conditions:
- L'utilisateur visualise l'historique des requêtes triées et filtrées selon ses besoins par chaîne de requêtes.

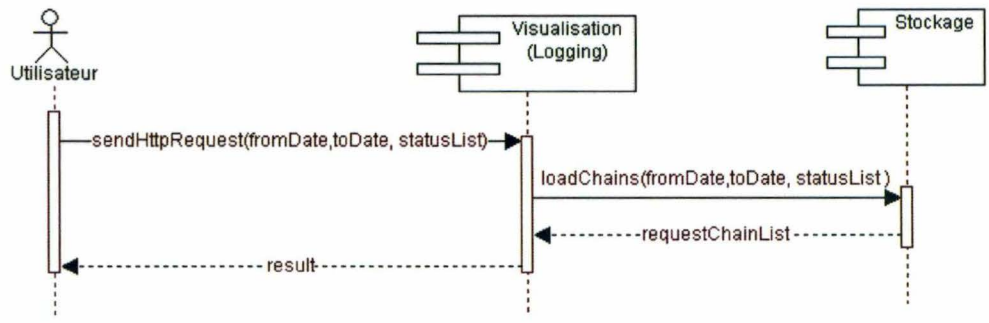



**Figure 42. Conception logique: use case de visualisation du logging par chaîne de requêtes.**

**sendHttpRequest**

- Utilité: La méthode illustre l'émission de la requête http faite par l'utilisateur pour obtenir l'historique des requêtes selon les critères désirés.
- Paramètres: La date de départ (fromDate), la date de fin (toDate), et les statuts de requêtes (statusList).
- Pré-condition: L'utilisateur est prêt à soumettre sa requête http à l'aide de son browser Internet.
- Post-condition: La requête a été traitée, et l'écran de résultat est affiché à l'utilisateur.

**loadChains**

- Utilité: Chargement de l'historique des requêtes sous forme d'une liste de chaînes de requêtes.
- Paramètres: La date de départ (fromDate), la date de fin (toDate) et les statuts de requêtes (statusList).
- Pré-condition: L'utilisateur soumet sa requête http à l'aide de son browser Internet.
- Post-condition: L'historique des requêtes est retourné sous forme d'une liste chronologique de chaînes de requêtes.

***Cas alternatif: Les dates de début et de fin sont invalides.***

Description:
Cas où l'utilisateur spécifie une date de départ postérieure à la date de fin, ou a omis de spécifier au moins l'une des 2 dates.

Pré-conditions:

- BNM est démarré et fonctionne correctement.
- Le système fonctionne correctement.
- L'utilisateur est identifié par le Portail et a atteint la portlet de logging.

Post-conditions:

- L'utilisateur visualise un message l'invitant à entrer correctement les dates demandées.

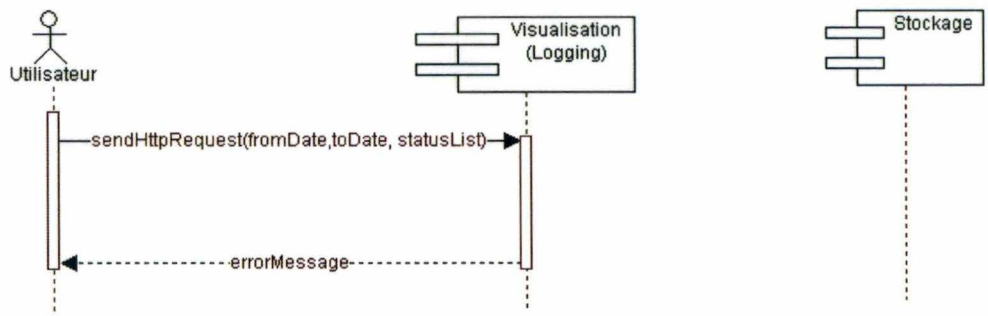



**Figure 43. Conception logique: use case de visualisation du logging par chaîne de requêtes en échec.**

## 6.3. La visualisation du monitoring

### VISUALISATION DU RESULTAT

Le résultat du monitoring se compose de 2 parties:
**Le monitoring des requêtes par composant**
Pour chaque composant, le monitoring utilise l'historique des requêtes reçues pour en déduire les compteurs suivants:

- le temps moyen de l'exécution d'une requête du composant considéré,
- le pourcentage de temps d'exécution pris par rapport au temps pris par les autres composants,
- le temps total d'exécution pris par les requêtes du composant considéré,
- le nombre de requêtes exécutées pour le composant considéré.

**Le monitoring des requêtes selon leur type** (cf. champ type de la classe Request en page 47)
Pour chaque type de requête, le monitoring utilise l'historique des requêtes reçues pour en déduire les compteurs suivants:

- le temps moyen de l'exécution d'une requête du type considéré,
- le pourcentage de temps d'exécution pris par rapport au temps pris par les autres types,
- le temps total d'exécution pris par les requêtes du type considéré,
- le nombre de requêtes exécutées pour le type considéré.

Vu que pour un composant, il existe plusieurs types de requêtes possibles, il est préférable d'afficher le résultat des requêtes selon leur type, groupées par composant.

## LA CLASSE MONITOREDELEMENT

La classe MonitoredElement a pour but de stocker les compteurs définis précédemment.
Chaque instance de cette classe représente une mesure des performances des objets de type Request pour un composant ou un type de requête considéré.
Le résultat du monitoring a donc la forme d'une liste d'objets de type MonitoredElement.

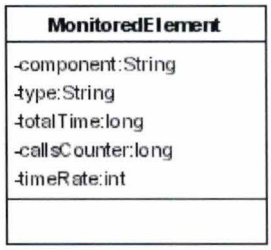



**Figure 44. Conception logique: la classe MonitoredElement.**

Les champs de la classe sont:

- **component** Le nom du composant commun à tous les objets de type Request dont les performances sont mesurées.
- **type** Le type de requête commun à tous les objets de type Request dont les performances sont mesurées. Ce champ n'a pas d'utilité dans le cas où la mesure est faite en fonction des composants uniquement.
- **totalTime** Le temps total d'exécution des requêtes obtenu en additionnant les temps d'exécution de chaque objet de type Request impliqué.
- **callsCounter** Le nombre total d'objets de type Request dont les performances sont mesurées.
- **timeRate** Le pourcentage du temps total (totalTime) par rapport aux autres objets de type MonitoredElement de la liste résultant du monitoring.

Remarque:
Il n'y a pas de champs pour le temps moyen de l'exécution d'une requête du type considéré car il se calcule directement par l'opération totalTime/callsCounter.

## LES FILTRES

Seul le filtre désignant l'intervalle de temps dans lequel les requêtes ont généré des traces dans le composant Stockage est nécessaire.

## USE CASE: VISUALISATION DU MONITORING PAR COMPOSANT

Description:
Ce diagramme de séquence illustre les interactions entre le composant Visualisation et Stockage lorsqu'un utilisateur demande à visualiser le monitoring par composant.

Pré-conditions:
- BNM est démarré et fonctionne correctement.
- Le système fonctionne correctement.
- L'utilisateur est identifié par le Portail et a atteint la portlet de monitoring.

Post-conditions:
- L'utilisateur visualise le monitoring des requêtes selon les composants.

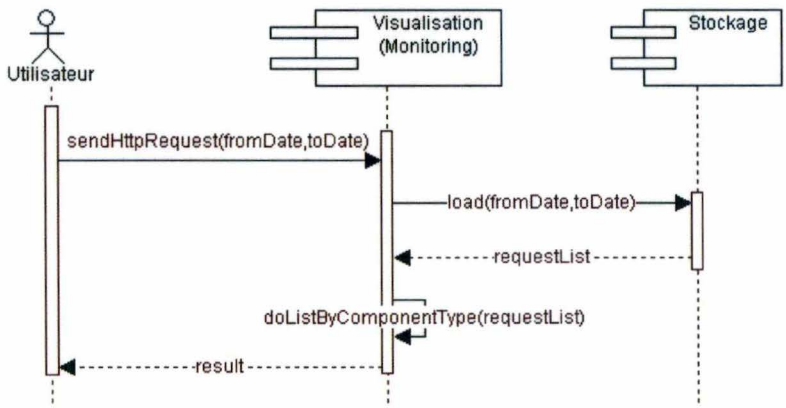



**Figure 45. Conception logique: use case de visualisation du monitoring par composant.**

**sendHttpRequest**
- Utilité: La méthode illustre l'émission de la requête http faite par l'utilisateur pour obtenir le monitoring des requêtes selon les critères désirés.
- Paramètres: La date de départ (fromDate) et la date de fin (toDate).
- Pré-condition: L'utilisateur est prêt à soumettre sa requête http à l'aide de son browser Internet.
- Post-condition: La requête a été traitée, et l'écran de résultat est affiché à l'utilisateur.

**load**
- Utilité: Chargement de l'historique des requêtes.
- Paramètres: La date de départ (fromDate) et la date de fin (toDate).
- Pré-condition: L'utilisateur a soumis sa requête http à l'aide de son browser Internet.
- Post-condition: L'historique des requêtes est retourné sous forme d'une liste chronologique d'objets de type Request.

**doListByComponentType(reqList)**
- Utilité: La méthode traite l'historique des chaînes de requêtes et en déduit la liste des objets de type MonitoredElement où chaque objet est le résultat du monitoring d'un composant.
- Paramètres: une liste d'objets de type Request.
- Pré-condition: L'utilisateur est prêt à soumettre sa requête http à l'aide de son browser Internet.
- Post-condition: La méthode retourne la liste d'objets de type MonitoredElement demandée.

***Cas alternatif: Les dates de début et de fin sont invalides.***

Description:
Cas où l'utilisateur spécifie une date de départ postérieure à la date de fin, ou a omis de spécifier au moins l'une des 2 dates.

Pré-conditions:
- BNM est démarré et fonctionne correctement.
- Le système fonctionne correctement.
- L'utilisateur est identifié par le Portail et a atteint la portlet de monitoring.

Post-conditions:
- L'utilisateur visualise un message l'invitant à entrer correctement les dates demandées.

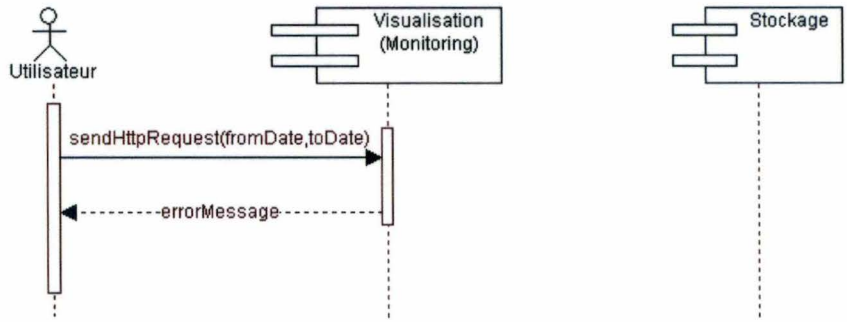



**Figure 46. Conception logique: use case de visualisation du monitoring par composant en échec.**

## USE CASE: VISUALISATION DU MONITORING PAR TYPE

Description:
Ce diagramme de séquence illustre les interactions entre le composant Visualisation et Stockage lorsqu'un utilisateur demande à visualiser le monitoring par type.

Pré-conditions:
- BNM est démarré et fonctionne correctement.
- Le système fonctionne correctement.
- L'utilisateur est identifié par le Portail et a atteint la portlet de monitoring.

Post-conditions:
- L'utilisateur visualise le monitoring des requêtes selon leur type.

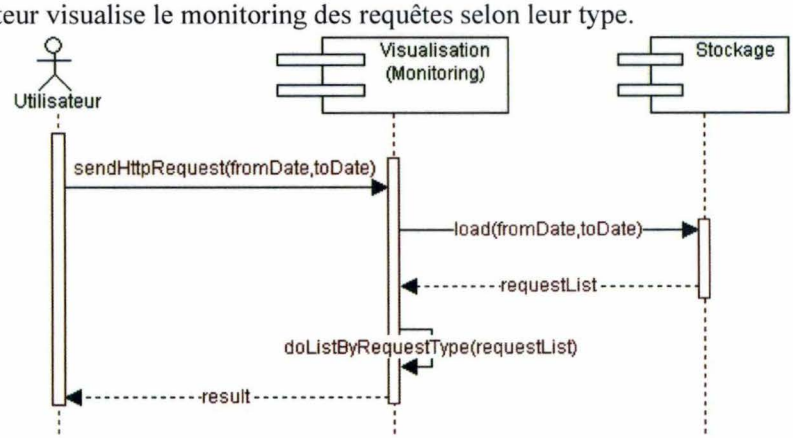



**Figure 47. Conception logique: use case de visualisation du monitoring par composant.**

**doListByRequestType**

- Utilité: La méthode traite l'historique des chaînes de requêtes et en déduit la liste des objets de type MonitoredElement où chaque objet est le résultat du monitoring d'un type de requête.
- Paramètres: Une liste d'objets de type Request.
- Pré-condition: L'utilisateur est prêt à soumettre sa requête http à l'aide de son browser Internet.
- Post-condition: La méthode retourne la liste d'objets de type MonitoredElement demandée.

***Cas alternatif: Les dates de début et de fin sont invalides.***

Ce cas alternatif est identique au cas alternatif du monitoring par composant en page 59.

## 6.4. La visualisation du benchmarking

### VISUALISATION DU RESULTAT

La visualisation du benchmarking est le résultat de la comparaison de 2 fichiers de monitoring (cf. Spec11 en page 43), où chaque fichier contient une liste d'objets de type MonitoredElement obtenu par un monitoring par type de requêtes.

La comparaison n'est effectuée que sur base du temps moyen de chaque MonitoredElement (totalTime/callsCounter).

Au final, l'utilisateur doit visualiser une liste où, par chaque type de requête, les temps moyens résultant des 2 monitorings sont affichés. Si un élément n'a pas d'équivalent dans l'autre fichier, il est affiché malgré tout avec un champ vide pour la valeur manquante.

### LA CLASSE BENCHMARKELEMENT

Cette classe sert à stocker les informations du benchmarking obtenues à partir de 2 objets de type MonitoredElement dont le type de requête est identique.
Le résultat du monitoring a donc la forme d'une liste d'objets de type BenchmarkElement.

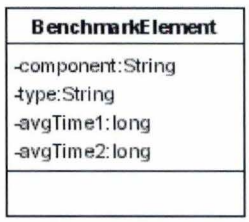



**Figure 48. Conception logique: la classe BenchmarkElement.**

Les champs de la classe sont:

- **component** Le nom du composant commun aux 2 objets de type MonitoredElement.
- **type** Le type de requête commun aux 2 objets de type MonitoredElement.
- **avgTime1** Le temps moyen d'exécution des requêtes du premier objet de type MonitoredElement.
- **avgTime2** Le temps moyen d'exécution des requêtes du second objet de type MonitoredElement.

Remarque:
Les 2 fichiers utilisés pour le benchmarking sont chargés et traités par le composant de Visualisation. Vu l'absence d'autres composants, il n'y a pas de use case utile à présenter.

# Section 7. Conception physique

Ce chapitre élabore l'architecture de l'application à l'échelle des packages et des classes, en tenant compte des contraintes technologiques.

Nous nous concentrons dans cette section exclusivement sur le composant Aspect qui est l'objet de ce mémoire et de ses interactions avec le composant Stockage.

## 7.1. La collecte des données

Les définitions des composants Aspect et Stockage peuvent être affinées à l'aide des critères déduits des pages précédentes et des choix d'implémentations:

- Un seul aspect suffit pour obtenir les données du logging et monitoring.
- L'aspect crée et utilise un objet de type Request pour gérer le cycle de vie de chaque requête.
- Log4j a été choisi comme gestionnaire de la source de stockage. Il a pour tâche de stocker les informations dans un fichier texte, ligne par ligne où chaque ligne est un nouveau log. Cela évite l'utilisation d'une base de données avec laquelle les échanges de requêtes engendreraient une charge supplémentaire pour l'application BNM.
- Le design pattern Data Access Object (DAO) est utilisé pour assurer la persistance des objets tout en facilitant d'éventuels changements de système de stockage.

Rappel à propos de JBoss-AOP:

- Tout advice d'un aspect est une méthode de classe ayant pour paramètre un objet dérivant du type Invocation. Dans notre cas le type est MethodInvocation car tous les advices à développer interceptentent des appels de méthode.
- Les poincuts et la relation advices-pointcuts sont à définir dans un fichier xml.

Remarque:
Le code source de l'aspect et son fichier de configuration xml sont disponibles en Annexe 2.

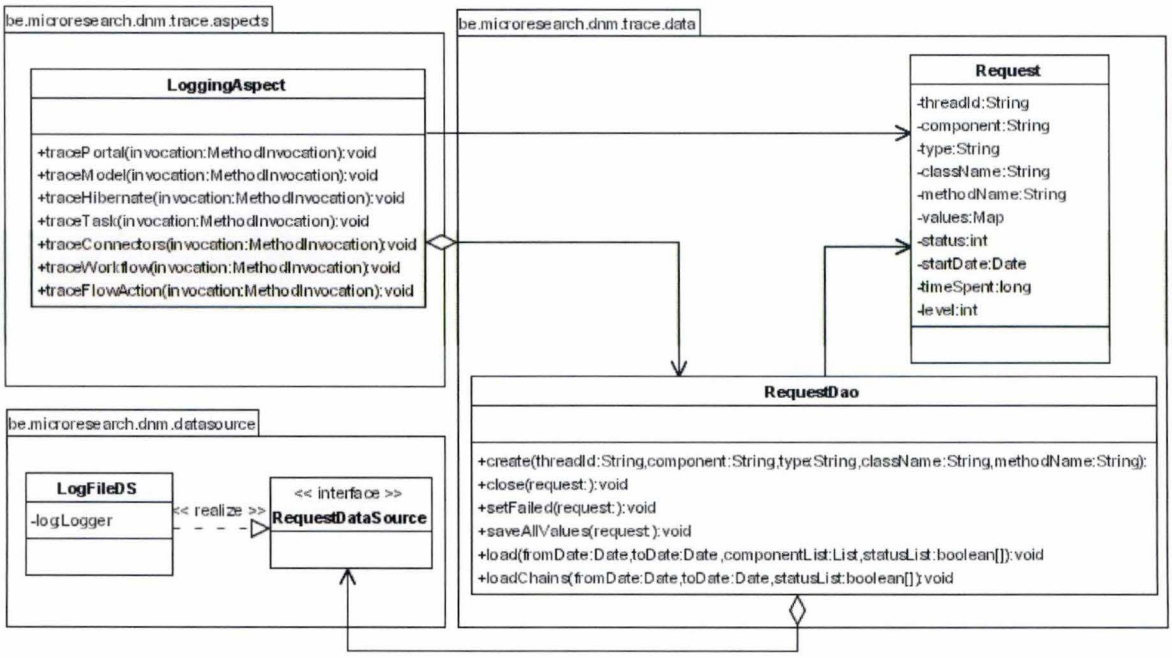



**Figure 49. Conception physique: diagramme de classes des composants Stockage et Aspect.**

Remarque: La dénomination BNM a été récemment adoptée pour des raisons de marketing. Par facilité, il a été décidé de conserver le nom initial de l'application DNM pour les packages.

**LoggingAspect**
La classe ayant le rôle d'aspect, contient tous les advices à exécuter pour assurer la gestion de traces. Chaque méthode qui y est déclarée correspond à un advice:

- **tracePortal** L'advice sur le Portail.
- **traceModel** L'advice sur le Modèle.
- **traceHibernate** L'advice sur le composant Hibernate.
- **traceTask** L'advice sur le Taskengine.
- **traceConnectors** L'advice sur le Connecteur.
- **traceWorkflow** L'advice sur le processus de Workflow.
- **traceWorkflowAction** L'advice sur toute action du Workflow.

**Request**
La classe est décrite en page 47.

**RequestDao**
LoggingAspect accède aux objets de type Request par l'intermédiaire de RequestDao. Cette classe a la charge d'assurer la persistance des données collectées. L'ensemble des méthodes sont définies dans les use cases de la Section 6.

**RequestDataSource**
Il s'agit de l'interface qui généralise tous les appels au système de gestion du stockage. Cette interface limite les modifications à apporter en cas de changement de source de stockage.

**LogFileDS**
L'implémentation de la source de stockage. La champ log est le Logger de log4j.

## USE CASE: TRAÇAGE D'UN JOINPOINT

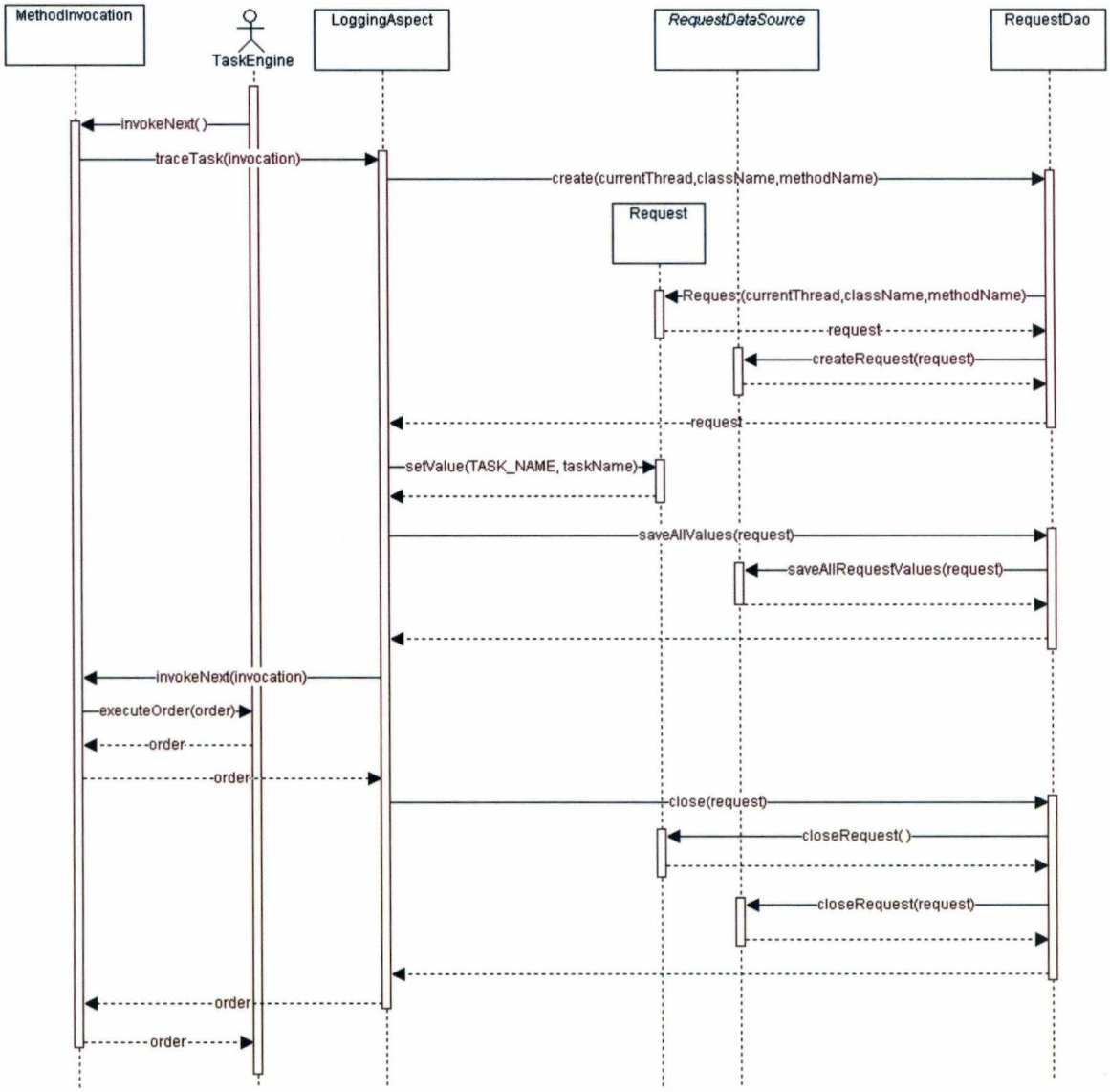
Description:
Ce diagramme de séquence illustre concrètement le déroulement du traçage d'une requête. Le joinpoint est défini sur le Taskengine, plus précisément sur l'exécution de la méthode executeOrder(order) de la classe Taskengine.

Pré-conditions:

- BNM est démarré et fonctionne correctement.
- Le système fonctionne correctement.
- L'utilisateur est identifié par le Portail.
- La requête de l'utilisateur est en cours, elle fait maintenant appel au Taskengine.

Post-conditions:

- La trace de la requête émise au Taskengine est sauvegardée.
- La requête utilisateur poursuit sont déroulement à travers les autres composants de l'application.




**invokeNext**

- Utilité: Cette méthode est insérée dans le code de la méthode ciblée par un pointcut lors de la phase d'instrumentation par JBoss-AOP. Son utilité est de déclencher l'interception de la méthode.
- Paramètres: Aucun.
- Pré-condition: Un poincut cible la méthode appelée.
- Post-condition: La méthode a été interceptée, l'advice et la méthode ont été exécutés et le résultat de la méthode interceptée est retourné (order).

**traceTask**

- Utilité: Il s'agit de l'advice de l'aspect dont le pointcut a été déclenché.
- Paramètres: L'objet détenant les informations sur la méthode appelée (invocation).
- Pré-condition: Un poincut lié à l'advice s'est déclenché.
- Post-condition: L'advice a généré un objet de type Request décrivant l'exécution de la méthode du Taskengine.

**create**

- Utilité: Création de l'instance de type Request et stockages des premières données collectées.

- Paramètres: L'identifiant de la thread en cours (currentThread), le nom de la classe interceptée (className) et le nom de la méthode interceptée (methodName).
- Pré-condition: Un advice est en cours d'exécution.
- Post-condition: L'objet de type Request est retourné et sa trace est sauvegardée.

**Request**
- Utilité: Constructeur de la classe Request. Il met son statut (status) en 'Started', et assigne l'heure de départ (dans startDate).
- Paramètres: L'identifiant de la thread en cours (currentThread), le nom de la classe interceptée (className) et le nom de la méthode interceptée (methodName).
- Pré-condition: Aucune.
- Post-condition: L'objet de type Request est retourné.

**createRequest**
- Utilité: Sauvegarde la trace de la création de la requête.
- Paramètres: L'objet de type Request.
- Pré-condition: Un objet de type Request a été instancié.
- Post-condition: La sauvegarde est effectuée.

**setValue**
- Utilité: Assigne une valeur spécifique à la méthode interceptée.
- Paramètres: Le nom de la valeur et la valeur.
- Pré-condition: Aucune.
- Post-condition: Aucune.

**saveAllValues**
- Utilité: Sauve les valeurs spécifiques à la requête exécutée.
- Paramètres: L'objet de type Request.
- Pré-condition: Au moins une valeur spécifique à la requête est collectée.
- Post-condition: La sauvegarde est effectuée.

**saveAllRequestValues**
Idem saveAllValues (car toute action sur l'objet de type Request passe par RequestDao).

**invokeNext**
- Utilité: Appeler l'intercepteur suivant de la chaîne ou la méthode interceptée s'il n'y a plus d'autres intercepteurs dans la chaîne.
- Paramètres: L'invocation (les informations sur la méthode interceptées).
- Pré-condition: L'advice a terminé les actions à effectuer avant l'exécution de la méthode interceptée.
- Post-condition: La méthode interceptée a terminé son exécution et son résultat (order) est retourné.

**executeOrder**
- Utilité: La méthode interceptée est exécutée.
- Paramètres: les informations sur la tâche à exécuter (order).
- Pré-condition: Aucun.
- Post-condition: La tâche est exécutée.

**close**
- Utilité: Clôture de la requête et sauvegarde de la trace.
- Paramètres: L'objet de type Request.
- Pré-condition: La requête a terminé son exécution.
- Post-condition: La sauvegarde est effectuée.

**closeRequest**

- Utilité: Clôture de la requête en assignant le temps total de l'exécution.
- Paramètres: Aucun.
- Pré-condition: La requête a terminé son exécution.
- Post-condition: L'objet de type Request a un statut Succeeded ou Failed.

**closeRequest**

- Utilité: Sauvegarde de la trace de clôture de la requête.
- Paramètres: L'objet de type Request.
- Pré-condition: L'objet de type Request a un statut Succeeded ou Failed.
- Post-condition: La sauvegarde est effectuée.

### *Cas alternatif: échec lors de l'exécution du joinpoint*

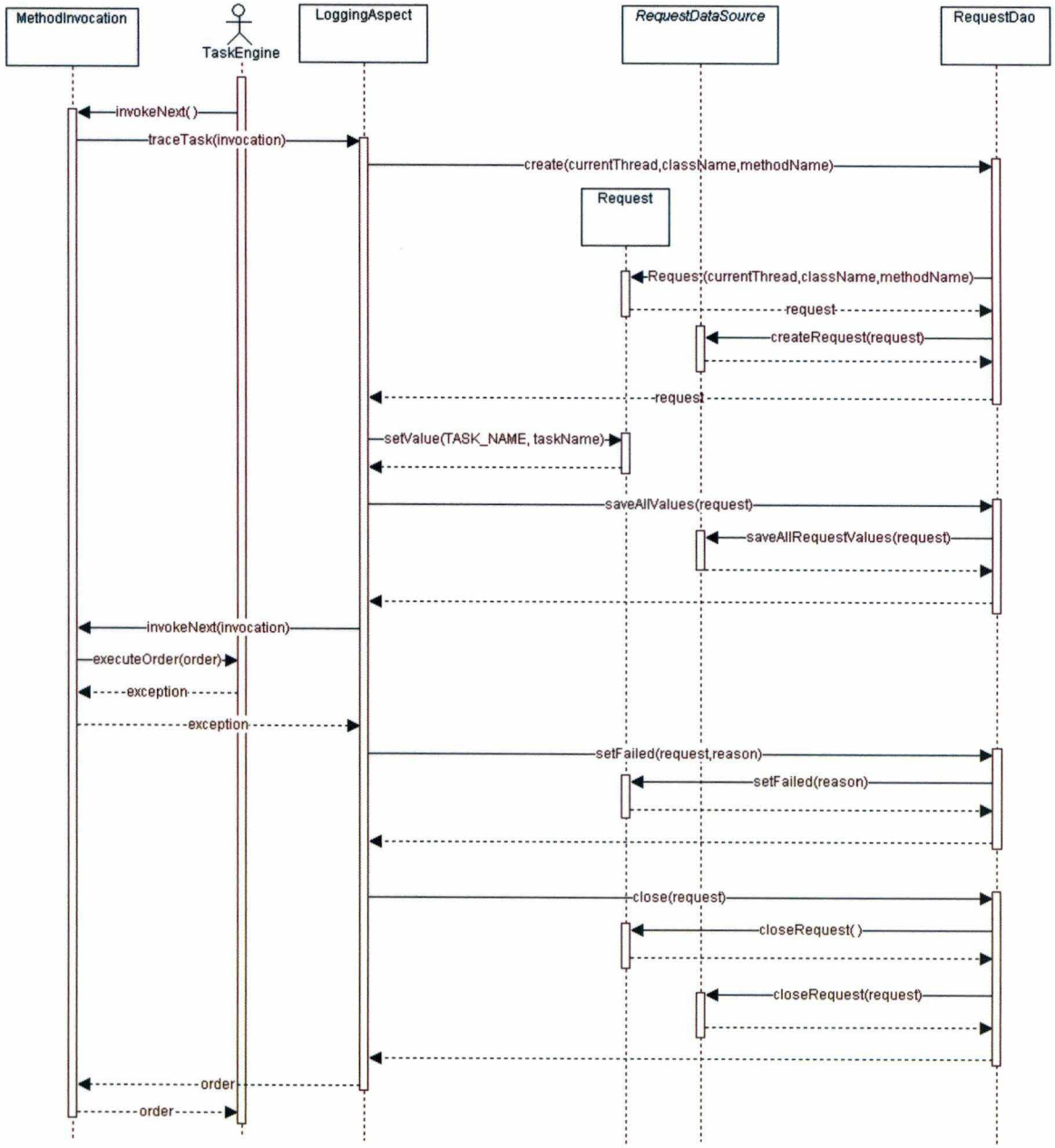
Description:
Le joinpoint défini sur le Taskengine retourne une Exception.

Pré-conditions:

- BNM est démarré et fonctionne correctement.
- Le système fonctionne correctement.
- L'utilisateur est identifié par le Portail.
- La requête de l'utilisateur est en cours, elle fait maintenant appel au Taskengine.

Post-conditions:

- La trace de la requête émise au Taskengine est stockée.
- La requête utilisateur est interrompue par l'erreur émise.




**setFailed**

- Utilité: Faire passer le statut de l'objet de type Request à Failed.
- Paramètres: L'objet de type Request et un éventuel message d'erreur.
- Pré-condition: Le statut de l'objet de type Request est Started.
- Post-condition: Le statut de l'objet de type Request est Failed.

**setFailed**

- Utilité: Faire passer le statut de l'objet de type Request à Failed.
- Paramètres: L'objet de type Request.
- Pré-condition: Le statut de l'objet de type Request est Started.
- Post-condition: Le statut de l'objet de type Request est Failed.

Remarque:
Toute action entraînant un changement d'état de l'objet de type Request passe par RequestDao.

# Section 8. Organisation des écrans de visualisation

Remarques:

- La phase de développement du système n'étant pas terminée, le lecteur peut constater quelques incohérences dans les captures d'écrans au niveau des résultats affichés.
- Pour la même raison, la spécification Spec11 (en page 43) prévoyant l'utilisation de couleurs pour afficher les valeurs comparées n'a pas encore été implémentée.

## 8.1. Visualisation du logging

La portlet de logging est composée de 3 écrans:

- le menu,
- le résultat du logging en fonction des requêtes,
- le résultat du logging en fonction des chaînes de requêtes.

### L'ECRAN DE MENU

Le menu donne la possibilité d'utiliser les différents filtres disponibles:

- Les dates début et de fin qui définissent l'intervalle de temps désiré. Le choix des dates se fait via un calendrier apparaissant sous forme de pop-up.
- Le choix des composants. Seules les requêtes appartenant aux composants sélectionnés seront visibles.
- Le choix des statuts. Seuls les requêtes ayant les statuts sélectionnés seront visibles.

L'utilisateur peut choisir la mise en forme du résultat:

- par requête,
- par chaîne de requêtes.

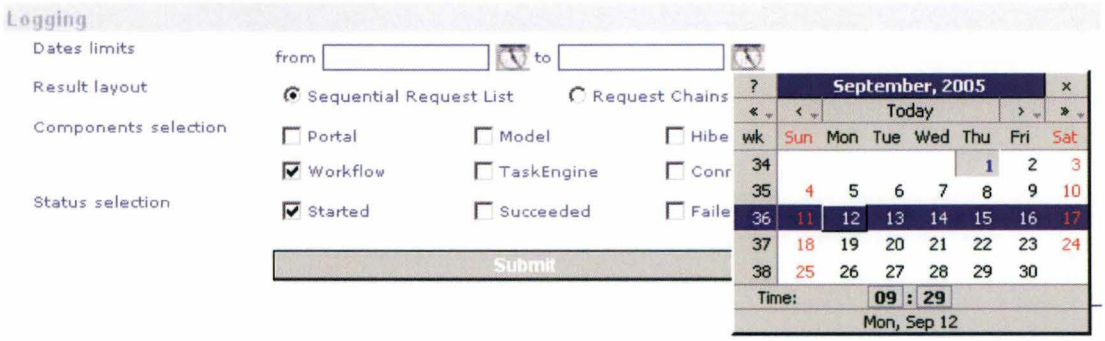
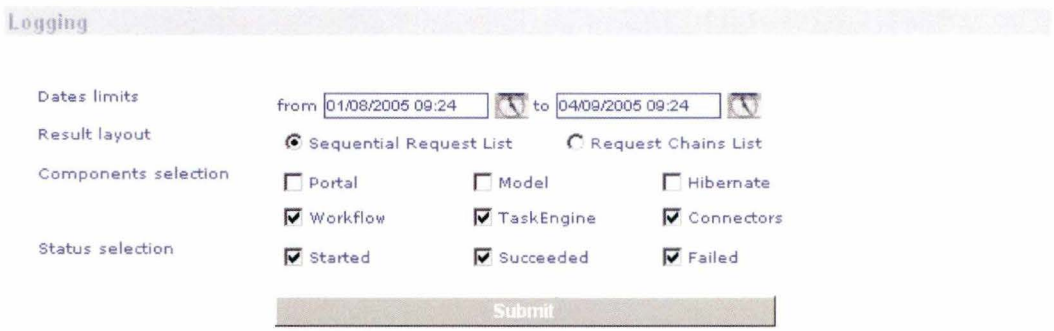



**Figure 50. Visualisation du logging: le menu.**

## L'ECRAN DE RESULTAT PAR REQUETE

L'écran de résultat est composé de:

- l'écran de menu,
- la liste des informations relatives aux requêtes exécutées durant l'intervalle de temps spécifié.




Tue Aug 30 11:48:09 CEST 2005

| Component | Type | Status | Time Spent |
|---|---|---|---|
| TaskEngine | ip.ConfigBGPOnPE | SUCCEEDED | 2937 |

Tue Aug 30 11:50:31 CEST 2005

| Component | Type | Status | Time Spent |
|---|---|---|---|
| WorkFlow Process | UNDEFINED | STARTED | 164033989 |

Tue Aug 30 11:50:31 CEST 2005

| Component | Type | Status | Time Spent |
|---|---|---|---|
| WorkFlow Process | ADSL | SUCCEEDED | 8360 |

Tue Aug 30 11:50:33 CEST 2005

| Component | Type | Status | Time Spent |
|---|---|---|---|
| TaskEngine | adsl.createDSLAM | SUCCEEDED | 4719 |

Tue Aug 30 11:50:36 CEST 2005

| Component | Type | Status | Time Spent |
|---|---|---|---|
| Connector | TL1Interaction | FAILED | 46 |
| **Other values** | | | |
| **REMOTE_HOST** localhost<br>**REMOTE_PORT** 13003 | | | |

Tue Aug 30 11:50:37 CEST 2005

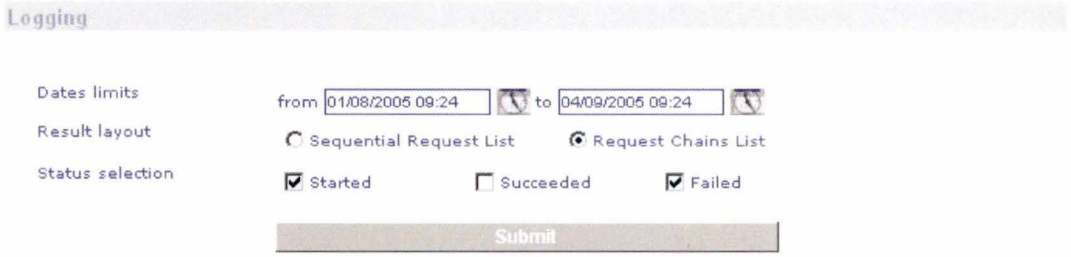
| Component | Type | Status | Time Spent |
|---|---|---|---|
| TaskEngine | adsl.createBAS | SUCCEEDED | 750 |

**Figure 51. Visualisation du logging: l'écran de résultat par requête.**

## L'ECRAN DE RESULTAT PAR CHAINE DE REQUETES

L'écran de résultat est composé de:

- l'écran de menu,
- la liste des informations relatives aux chaînes de requêtes exécutées durant l'intervalle de temps spécifié.




Tue Aug 30 11:50:31 CEST 2005

| Component | Type | Time Spent | Status |
|---|---|---|---|
| WorkFlow Process | UNDEFINED | 164172510 | STARTED |
| WorkFlow Process | ADSL | 8360 | SUCCEEDED |
| Hibernate | SQL | 0 | SUCCEEDED |
| TaskEngine | adsl.createDSLAM | 4719 | SUCCEEDED |
| Hibernate | SQL | 0 | SUCCEEDED |
| Hibernate | SQL | 0 | SUCCEEDED |
| Hibernate | SQL | 0 | SUCCEEDED |
| Hibernate | SQL | 0 | SUCCEEDED |
| Hibernate | SQL | 0 | SUCCEEDED |
| Connector | TL1Interaction | 46 | FAILED |
| Hibernate | SQL | 0 | SUCCEEDED |
| Hibernate | SQL | 125 | SUCCEEDED |
| Hibernate | SQL | 0 | SUCCEEDED |
| TaskEngine | adsl.createBAS | 750 | SUCCEEDED |
| Hibernate | SQL | 0 | SUCCEEDED |
| Hibernate | SQL | 0 | SUCCEEDED |
| WorkFlow Action | ADSL | 62 | SUCCEEDED |
| WorkFlow Process | UNDEFINED | 163967371 | STARTED |
| WorkFlow Process | ADSL | 422 | SUCCEEDED |
| WorkFlow Action | ADSL | 16 | SUCCEEDED |

Tue Aug 30 11:53:29 CEST 2005

| Component | Type | Time Spent | Status |
|---|---|---|---|
| WorkFlow Process | UNDEFINED | 163994685 | STARTED |
| WorkFlow Process | ADSL | 828 | SUCCEEDED |
| Hibernate | SQL | 0 | SUCCEEDED |
| TaskEngine | adsl.createDSLAM | 219 | SUCCEEDED |
| Hibernate | SQL | 0 | SUCCEEDED |
| Hibernate | SQL | 0 | SUCCEEDED |
| Hibernate | SQL | 0 | SUCCEEDED |
| Connector | TL1Interaction | 0 | SUCCEEDED |
| Hibernate | SQL | 0 | SUCCEEDED |
| Hibernate | SQL | 0 | SUCCEEDED |
| Hibernate | SQL | 0 | SUCCEEDED |
| TaskEngine | adsl.createBAS | 47 | SUCCEEDED |
| Hibernate | SQL | 0 | SUCCEEDED |
| Hibernate | SQL | 0 | SUCCEEDED |
| WorkFlow Action | ADSL | 0 | SUCCEEDED |

**Figure 52. Visualisation du logging: l'écran de résultat par chaîne de requêtes.**

## 8.2. Visualisation du monitoring

La portlet de logging est composée de 3 écrans:

- le menu,
- le résultat du monitoring en fonction des composants,
- le résultat du monitoring en fonction des types de requête.

## L'ECRAN DE MENU

Le menu permet d'entrer les dates début et de fin définissant l'intervalle de temps désiré. Le choix des dates se fait via un calendrier apparaissant sous forme de pop-up.

L'utilisateur peut choisir la mise en forme du résultat:

- par composant,
- par type de requêtes.

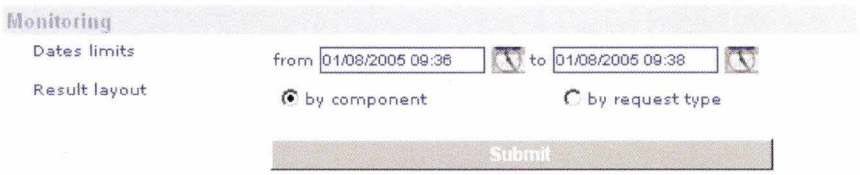
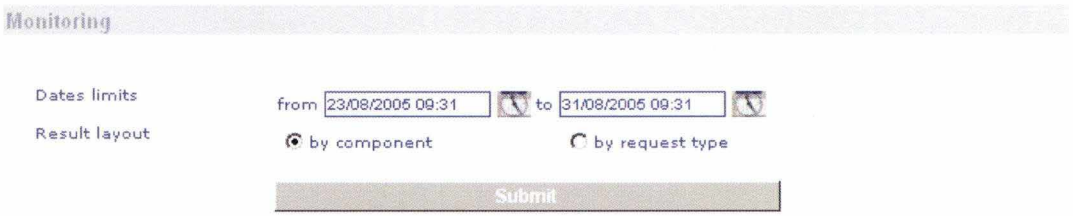



**Figure 53. Visualisation du monitoring: le menu.**

## L'ECRAN DE RESULTAT PAR COMPOSANT

L'écran de résultat est composé de:

- l'écran de menu,
- une option pour sauvegarder le résultat du monitoring dans un fichier texte sur son PC,
- la liste des informations relatives au monitoring des composants pour les requêtes exécutées durant l'intervalle de temps spécifié.




Save the monitoring result locally.

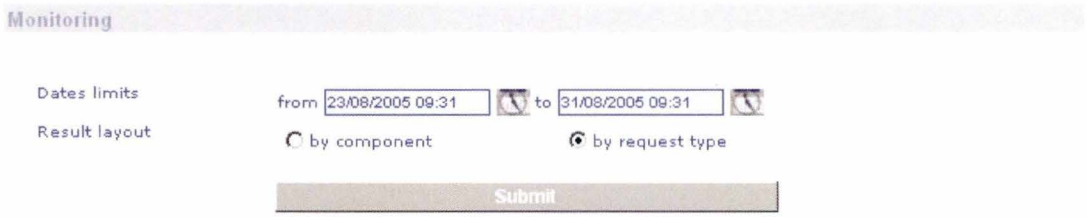
| Component | Average time | Time rate | Total time | Nb of calls |
|---|---|---|---|---|
| WorkFlow Process | 82177379 millisec | 99% | 493064279 millisec | 6 |
| Model | 113 millisec | 0% | 797 millisec | 7 |
| Portal | 406 millisec | 0% | 4874 millisec | 12 |
| WorkFlow Action | 26 millisec | 0% | 78 millisec | 3 |
| Hibernate | 2 millisec | 0% | 266 millisec | 99 |
| Connector | 23 millisec | 0% | 46 millisec | 2 |
| TaskEngine | 1734 millisec | 0% | 8672 millisec | 5 |

**Figure 54. Visualisation du monitoring: l'écran de résultat par composant.**

### L'ECRAN DE RESULTAT PAR TYPE DE REQUETE

L'écran de résultat est composé de:

- l'écran de menu,
- une option pour sauvegarder le résultat du monitoring dans un fichier texte sur son PC,
- la liste des informations relatives au monitoring en fonction des types de requêtes pour l'intervalle de temps spécifié.




Save the monitoring result locally.

**WorkFlow Process**

| Type | Average time | Time rate | Total time | Nb of calls |
|---|---|---|---|---|
| ADSL | 1614 millisec | 0% | 9688 millisec | 6 |
| UNDEFINED | 164578223 millisec | 99% | 493734669 millisec | 3 |

**Model**

| Type | Average time | Time rate | Total time | Nb of calls |
|---|---|---|---|---|
| findAllNetworkElementTypes | 390 millisec | 0% | 390 millisec | 1 |
| findAllConnectorDefinition | 62 millisec | 0% | 62 millisec | 1 |
| ejbCreate | 16 millisec | 0% | 16 millisec | 1 |
| findAllEquipmentHolderTypes | 94 millisec | 0% | 94 millisec | 1 |
| getNetworkElementWithEquipmentHolders | 78 millisec | 0% | 78 millisec | 1 |
| setSessionContext | 0 millisec | 0% | 0 millisec | 1 |
| find | 157 millisec | 0% | 157 millisec | 1 |

**Portal**

| Type | Average time | Time rate | Total time | Nb of calls |
|---|---|---|---|---|
| Portal | 406 millisec | 0% | 4874 millisec | 12 |

**Hibernate**

| Type | Average time | Time rate | Total time | Nb of calls |
|---|---|---|---|---|
| SQL | 2 millisec | 0% | 266 millisec | 99 |

**Connector**

| Type | Average time | Time rate | Total time | Nb of calls |
|---|---|---|---|---|
| TL1Interaction | 23 millisec | 0% | 46 millisec | 2 |

**TaskEngine**

| Type | Average time | Time rate | Total time | Nb of calls |
|---|---|---|---|---|
| ip.ConfigBGPOnPE | 2937 millisec | 0% | 2937 millisec | 1 |
| adsl.createDSLAM | 2469 millisec | 0% | 4938 millisec | 2 |
| adsl.createBAS | 398 millisec | 0% | 797 millisec | 2 |

**Figure 55. Visualisation du monitoring: l'écran de résultat par type de requête.**

## 8.3. Visualisation du benchmarking

La portlet de logging est composée de 2 écrans:

- le menu,
- le résultat du benchmarking.

## L'ECRAN DE MENU

Le menu permet de sélectionner les 2 fichiers obtenus à partir d'un des écrans de résultat du monitoring.

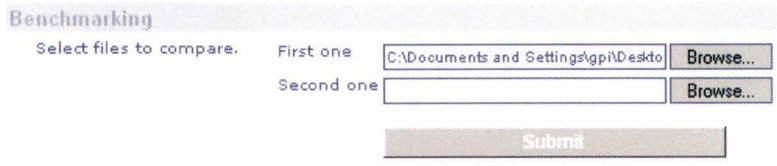



**Figure 56. Visualisation du benchmarking: le menu.**

## L'ECRAN DE RESULTAT

L'écran de résultat est composé de:

- l'écran de menu,
- la liste des informations relatives au benchmarking.

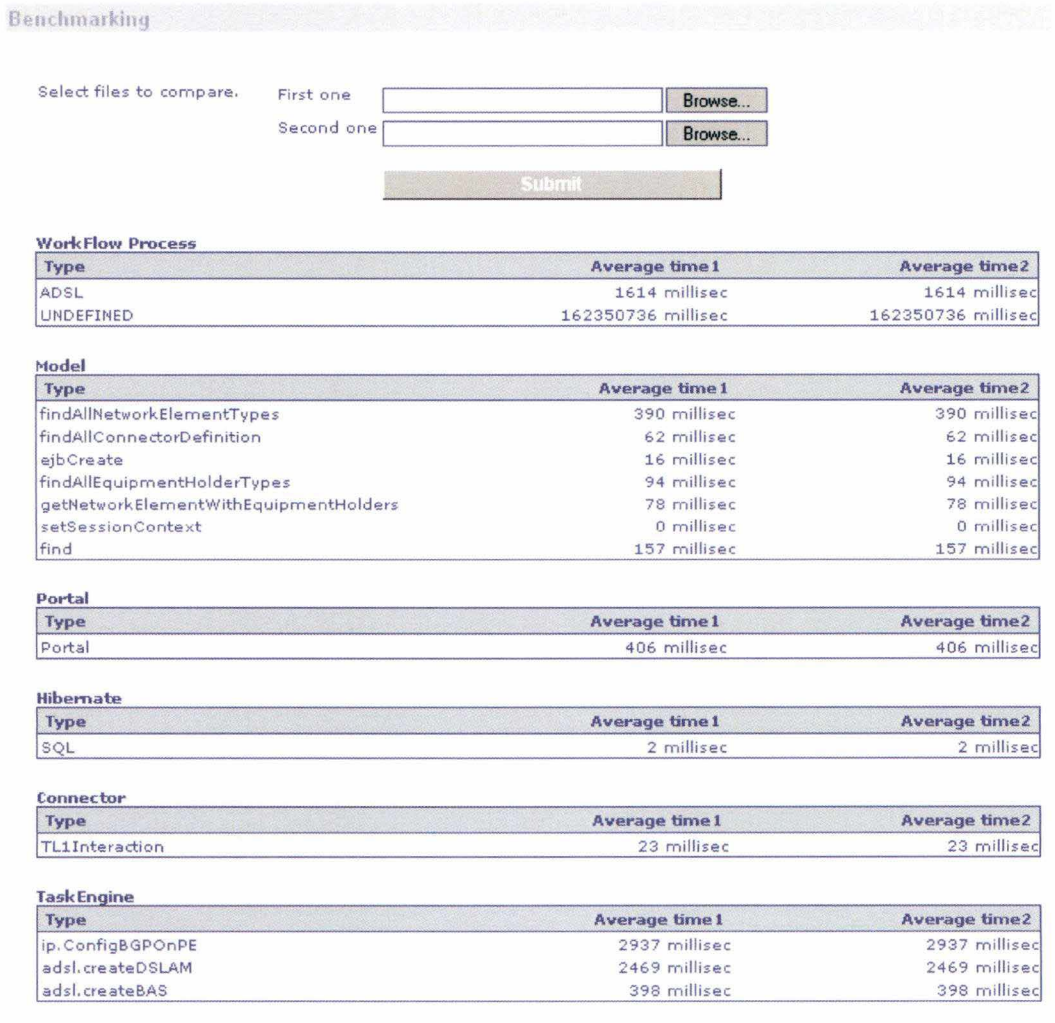



**Figure 57. Visualisation du benchmarking: l'écran de résultat.**

# Conclusions

## Les concepts

Le paradigme de programmation orienté aspect rend les composants plus modulaires en isolant le code source des préoccupations transversales dans des aspects. Le résultat est une application dont le code source ne contient aucune ligne de code relative aux préoccupations ainsi isolées. C'est au tissage que les relations entre application et aspects sont établies, sur base des pointcuts déclarés dans les aspects.

De ce fait, l'ajout, la modification ou la suppression d'un aspect ne nécessite aucune modification du code source de l'application.

Par contre, si le code source de l'application est modifié, il est possible que le changement ait un effet sur le fonctionnement de l'aspect. Cela s'explique par les pointcuts, dont le rôle est de désigner un (des) endroit(s) du code source de l'application sur base du nom de ses packages, classes, variables, méthodes, et arguments des méthodes.

## Les outils

Trois outils ont été évalués.

- Le premier, **AspectJ**, est un tisseur statique performant. Il s'avère être un bon choix dans le cadre d'une application dont l'arrêt ne constitue pas un évènement critique.
  Son plug-in Eclipse fournit tous les outils utiles pour programmer efficacement.
- **JBoss-AOP** a été choisi pour le développement du projet. Son tissage dynamique fournit le moyen d'apporter des modifications aux aspects sans devoir recompiler l'application, ni l'arrêter. Cette caractéristique est attrayante pour les compagnies devant assurer un service constant.
  De plus, le serveur d'applications JBoss permet à l'aide de sa console web de vérifier la validité des poincuts configurés quand l'application est déployée sur le serveur.
- **Spring AOP** a une conception comparable à JBoss-AOP. Cependant, il ne fournit aucun outil permettant de vérifier la validité du tissage. De plus, son utilisation en dehors du serveur d'applications Spring oblige d'adapter le code source de l'application en conséquence.

## L'analyse, la conception et le développement

Pour la phase d'analyse, la connaissance du domaine d'application nécessite l'étude du code source des composants de l'application et des autres librairies utilisées. Si AOP permet de ne pas toucher au code source de l'application, il faut néanmoins en disposer et le comprendre pour pouvoir y tisser un aspect.

Côté conception, on constate qu'UML ne définit pas la notion d'aspect. L'analyste a donc la liberté de définir un type de représentation ou de le choisir parmi les profiles UML pour AOP existants.

La phase de développement a mis en évidence la difficulté de déclarer les pointcuts. D'une part, les erreurs de syntaxe dans les expressions définissant les pointcuts sont vite arrivées, et d'autre part, il faut pouvoir s'assurer que tout pointcut créé pointe bien vers les endroits du code souhaité. Cette difficulté aurait été nettement moindre si le plug-in de JBoss-AOP pour Eclipse, avait fonctionné correctement (la fenêtre de l'éditeur devant montrer les effets du tissage est restée désespérément

vide). Heureusement, il fut néanmoins possible de visualiser l'effet du tissage des aspects sur l'application déployée via la web-console du serveur d'applications.

Ce dernier point souligne à nouveau le risque engendré par l'utilisation des aspects.
Pour l'évolution et la maintenance de l'application, il est prioritaire que tout développeur travaillant sur l'application ait connaissance des aspects configurés. La raison n'est pas l'incidence de l'aspect sur l'application, car la préoccupation transversale qu'il résout est relativement indépendante. Le risque provient surtout des modifications faites dans l'application, qui risque d'engendrer des problèmes multiples sur l'aspect.

Trois solutions peuvent être envisagées pour ce problème:
- La documentation technique de l'application doit mentionner où les pointcuts agissent.
- Tous les développeurs de l'application doivent utiliser un IDE muni du plug-in requis (fonctionnant correctement). Ceci n'est valable que si les développeurs disposent des sources des aspects utilisés sous peine de ne rien visualiser de particulier.
- Marquer les endroits du code source de l'application visés par des pointcuts par un commentaire. Au-delà de la question de la préservation de l'indépendance du code source de l'application, il s'avère difficile d'appliquer cette méthode si le poincut impacte des centaines de méthodes disséminées dans des dizaines de classes.

## Le résultat du projet

Un aspect a été utilisé pour effectuer la collecte de données au sein de l'application.

L'application a pu rester indépendante de cette préoccupation car elle est isolée dans un fichier séparé. L'avantage est qu'à chaque fois qu'une nouvelle version de ce fichier est disponible, il est déployé sur le serveur d'applications et tissé dynamiquement à l'application qui ne cesse à aucun moment son activité. De nouvelles valeurs peuvent ainsi être collectées dans l'application sans que la modification n'entrave son fonctionnement. Cette facilité apporte une valeur ajoutée aux utilisateurs de l'application qui ne sont pas interrompus dans leur travail.

Si l'ensemble du code source de la préoccupation relative au logging, monitoring et benchmarking est générique, ce n'est pas le cas de l'aspect responsable de la collecte des données. En effet, l'aspect a été conçu pour agir à des endroits spécifiques du code source de l'application, et ses advices pour collecter des données spécifiques de l'application.

L'aspect développé doit donc être considéré comme un composant dont le rôle est d'effectuer le branchement du composant de logging, monitoring et benchmarking sur l'application voulue. L'aspect doit être adapté pour être tissé à une autre application, mais laisse le code source de l'application indépendante.

## Autres observations

Il semble qu'actuellement la concrétisation la plus marquante du paradigme de programmation orienté aspect soit dans le monde des serveurs d'applications. Leur évolution actuelle permet d'entrevoir dans un avenir proche des applications ne contenant plus que la logique métier où les services désirés (transaction, cache, authentification, ...) seront choisis parmi ceux disponibles et ajoutés via des aspects.
Une belle avancée pour la réutilisation des composants...

Il est intéressant de souligner que ces serveurs d'applications offrent leur propre solution AOP dans une optique purement objet, avec des proxies et des intercepteurs. Ce choix facilite le débugging de

l'application et des aspects. En effet, la recherche d'erreurs ne pose aucun problème via un débuggeur ou l'état de la pile d'exécution[7].

## Les limites du travail

Le paradigme de programmation orienté aspect a été abordé sous de nombreux angles.

Dans le premier chapitre, les concepts généraux ont été présentés, ainsi que les techniques de tissage, l'évaluation de quelques outils existants pour le monde Java et enfin un bref comparatif aux design patterns.

Dans le second chapitre, c'est un cas réel de développement d'un projet utilisant un aspect qui est illustré au travers de son analyse et sa conception.

Cette approche a le mérite de donner une vue d'ensemble sur le paradigme. Cependant certains points n'ont pas été étudiés en profondeur:

- Seuls les comportements statiques des aspects ont vraiment été étudiés. Certains comportements dynamiques tels que les dynamic control flows ont été cités, mais jamais vraiment testés ni utilisés.
- La partie relatant des design patterns comme solution alternative (cf. Annexe 1 en page 76) avait principalement pour but de faire le parallélisme entre les 2 types de conception. Cependant il existe des travaux traitant des design patterns où chaque pattern du "Gang of Four" est implémenté en orienté objet et en orienté aspect afin de comparer la qualité de chacun en terme de modularité (par exemple: [Hannemann et al. 02]).
- Le projet a mené à l'implémentation d'un aspect seul. Aucune conclusion n'a pu donc être tirée sur un développement impliquant de multiples aspects et leurs interactions.

## L'aboutissement

Ce travail représente une expérience enrichissante.

En plus d'AOP, certaines facettes du paradigme objet, des design patterns, de l'analyse et de la conception me paraissent maintenant beaucoup plus claires.

Des barrières psychologiques ont également été forcées par la démystification du code des applications Open source: étudier directement leur code apporte finalement des réponses bien plus claires que de nombreux livres ou tutoriels.

Tout cela remplit grandement l'objectif de renforcer des connaissances utiles à mon orientation et mes aspirations professionnelles.

[7] Cette information est obtenue à partir d'une Exception Java, avec la méthode printStackTrace( ) bien connue des programmeurs.

# Annexe 1. Une solution alternative: les design patterns

Il existe des techniques de programmation éprouvées, appelées patrons de conception ou design patterns. Ceux-ci montrent comment utiliser au mieux les mécanismes de l'orienté objet pour améliorer, entre autres, l'architecture des systèmes à développer.

Le nombre de design patterns existants est important, et la plupart apportent une amélioration en termes de modularité, cela à des degrés divers. Prenons le "Décorateur" à titre d'exemple (voir Figure 58).

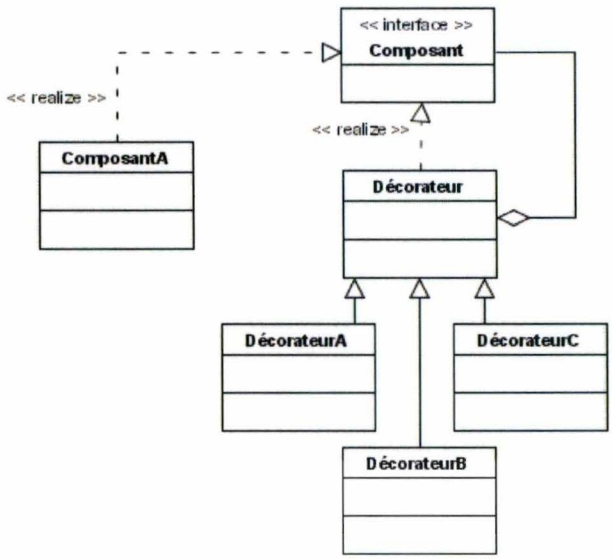



**Figure 58.Le diagramme de classes du design pattern Décorateur.**

Il fait partie des incontournables patterns du "Gang of Four" [Gamma et al. 95]. Son but est de joindre dynamiquement de nouvelles propriétés à des objets dérivant d'un même type.

Durant l'exécution, tout instance de `Composant` peut être enrichie par les fonctionnalités des classes dérivant du type `Décorateur`.
Cette architecture fournit des comportements particuliers isolés dans des classes, des comportements pouvant être modifiés par d'autres classes. Si l'on considère les classes dérivant du type `Décorateur` comme étant des préoccupations distinctes, on retrouve les concepts de l'isolement du code source des préoccupations, et leur mise en relation avec le composant principal à l'exécution.

Cet exemple montre qu'effectivement l'utilisation des design patterns offre des solutions pour un gain de modularité. La principale caractéristique que l'on retrouve généralement dans ces patterns, c'est l'obligation d'uniformiser toutes les préoccupations selon une interface ou une classe abstraite commune. Elle seule est déclarée et utilisée dans le code source de l'application.

La limite de ce type de solution est donc que le design du système doit être conçu au préalable avec le(s) design pattern(s) adapté(s) aux préoccupations susceptibles de changer. De plus, toute

modification apportée ultérieurement tel que l'ajout d'une préoccupation doit obligatoirement respecter l'architecture imposée par le design pattern implémenté dans le système.

# Annexe 2. Fichiers source et de configuration de l'aspect.

## FICHIER JBOSS-AOP.XML

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<aop>

<aspect class="be.microresearch.dnm.trace.aspects.LoggingAspect" scope="PER_VM"/>

<!-- on Portal -->
<pointcut name="ActionMethod"
          expr="execution(public * com.mr.dnm.*.action.*Action-&gt;execute(..))"/>
<bind pointcut="ActionMethod">
       <advice aspect="be.microresearch.dnm.trace.aspects.LoggingAspect" name="tracePortal"/>
</bind>

<!-- on DNM model -->
<pointcut name="ModelEjbMethod" expr="execution(public * com.mr.dnm.ejb.*Bean-&gt;*(..))"/>
<bind pointcut="ModelEjbMethod">
       <advice aspect="be.microresearch.dnm.trace.aspects.LoggingAspect" name="traceModel"/>
</bind>

<!-- on Hibernate (V3.0) -->
<pointcut name="logSqlMethod" expr="execution(private void org.hibernate.jdbc.*-
&gt;log(..))"/>
<bind pointcut="logSqlMethod">
       <advice aspect="be.microresearch.dnm.trace.aspects.LoggingAspect"
               name="traceHibernate"/>
</bind>

<!-- on Workflow (JBPM V2.0-beta2) -->
<pointcut name="jbpmStartProcessInstanceMethod"
       expr="execution(public * org.jbpm.impl.ExecutionServiceImpl-
&gt;startProcessInstance(java.lang.String,java.lang.Long,java.util.Map,java.lang.String))"/>
<bind pointcut="jbpmStartProcessInstanceMethod">
       <advice aspect="be.microresearch.dnm.trace.aspects.LoggingAspect"
               name="traceWorkflow"/>
</bind>

<pointcut name="jbpmExecuteMethod" expr="execution(public void
org.jbpm.model.definition.impl.ActionImpl-&gt;execute(org.jbpm.impl.ExecutionContextImpl))"/>
<bind pointcut="jbpmExecuteMethod">
       <advice aspect="be.microresearch.dnm.trace.aspects.LoggingAspect"
               name="traceFlowAction"/>
</bind>

<!-- on TaskEngine -->
<pointcut name="taskengineExecuteOrderMethod" expr="execution(public com.mr.task.Order
com.mr.task.TaskEngine-&gt;executeOrder(com.mr.task.Order))"/>
<bind pointcut="taskengineExecuteOrderMethod">
       <advice aspect="be.microresearch.dnm.trace.aspects.LoggingAspect" name="traceTask"/>
</bind>

<!-- on Connectors -->
<pointcut name="connectorsMethod" expr="execution(private *
                          be.microresearch.commonra.ra.CommonInteraction-&gt;exec(..))"/>
<bind pointcut="connectorsMethod">
       <advice aspect="be.microresearch.dnm.trace.aspects.LoggingAspect"
               name="traceConnectors"/>
</bind>

</aop>
```

## FICHIER LOGGINGASPECT.JAVA

```
/*
 * Created on 18-mai-2005
 */
package be.microresearch.dnm.trace.aspects;

import java.lang.reflect.Method;

import javax.servlet.http.HttpServletRequest;

import org.apache.log4j.Logger;
import org.jboss.aop.joinpoint.ConstructorInvocation;
import org.jboss.aop.joinpoint.MethodInvocation;
import org.jbpm.impl.ExecutionContextImpl;
import org.jbpm.impl.ExecutionServiceImpl;
import org.jbpm.model.definition.Definition;

import com.mr.task.Order;

import be.microresearch.commonra.ra.CommonInteraction;
import be.microresearch.commonra.ra.CommonManagedConnectionFactory;
import be.microresearch.dnm.trace.data.Request;
import be.microresearch.dnm.trace.data.RequestDao;
import be.microresearch.dnm.trace.utils.ReferenceNames;


/**
 * BNM aspect.
 * @author gpi
 */
public class LoggingAspect {

    private Logger logger = Logger.getLogger(this.getClass());
    private RequestDao reqDao = RequestDao.getInstance();


    /**
     * Advice on the Portal pointcut.
     * @param invocation
     * @return
     * @throws Throwable
     */
    public Object tracePortal(MethodInvocation invocation) throws Throwable {

        Request req = createRequest(invocation, ReferenceNames.PORTAL, ReferenceNames.PORTAL);
        try {
            Object[] args = invocation.getArguments();
            if ((args!=null) && (args[0]!=null)) {
                for (int i=0; i<args.length; i++) {
                    int a = args[i].getClass().toString().indexOf("Request");
                    if (args[i].getClass().toString().indexOf("Request")>0) {
                        HttpServletRequest httpReq = (HttpServletRequest)args[i];
                        req.setValue(ReferenceNames.USER_NAME, httpReq.getRemoteUser());
                    }
                }
            }
            reqDao.saveAllValues(req);
            return invocation.invokeNext();
        } catch (Exception e) {
            reqDao.setFailed(req, checkMsgNotEmpty(e));
            throw e;
        } finally {
            reqDao.close(req);
        }
    }
```

```
    /**
     * Advice on the DNM-Model EJB.
     * @param invocation
     * @return
     * @throws Throwable
     */
    public Object traceModel(MethodInvocation invocation) throws Throwable {

        Request req = createRequest(invocation, ReferenceNames.MODEL,
invocation.getMethod().getName());
        try {
            return invocation.invokeNext();
        } catch (Exception e) {
            reqDao.setFailed(req, checkMsgNotEmpty(e));
            throw e;
        } finally {
            reqDao.close(req);
        }
    }


    /**
     * Advice on the Hibernate pointcut.
     * @param invocation
     * @return
     * @throws Throwable
     */
    public Object traceHibernate(MethodInvocation invocation) throws Throwable {

        Request req = createRequest(invocation, ReferenceNames.HIBERNATE,
ReferenceNames.SQL_REQUEST);
        try {
            Object[] args = invocation.getArguments();
            if ((args!=null) && (args[0]!=null)) {
                String sqlRequest = (String)args[0];
                req.setValue(ReferenceNames.HIBERNATE_SQL_TYPE, sqlRequest);
            }
            reqDao.saveAllValues(req);
            return invocation.invokeNext();
        } catch (Exception e) {
            reqDao.setFailed(req, checkMsgNotEmpty(e));
            throw e;
        } finally {
            reqDao.close(req);
        }
    }


    /**
     * Advice on the TaskEngine pointcut.
     * @param invocation
     * @return
     * @throws Throwable
     */
    public Object traceTask(MethodInvocation invocation) throws Throwable {
        Request req = null;
        try {

            Method method = invocation.getMethod();

            Object[] args = invocation.getArguments();
            if ((args!=null) && (args[0]!=null)) {
                for (int i=0; i<args.length; i++) {
                    int a = args[i].getClass().toString().indexOf("Request");
                    if (args[i].getClass().toString().indexOf("Order")>0) {
                        Order order = (Order)args[i];
                        req = createRequest(invocation, ReferenceNames.TASK_ENGINE,
order.getTaskName());
                        break;
                    }
                    req = createRequest(invocation, ReferenceNames.TASK_ENGINE,
ReferenceNames.UNDEFINED);
                }
            } else {
                req = createRequest(invocation, ReferenceNames.TASK_ENGINE,
ReferenceNames.UNDEFINED);
            }
```

```
            reqDao.saveAllValues(req);
            return invocation.invokeNext();
        } catch (Exception e) {
            reqDao.setFailed(req, checkMsgNotEmpty(e));
            throw e;
        } finally {
            reqDao.close(req);
        }
    }


    /**
     * Advice on the Connectors pointcut.
     * @param invocation
     * @return
     * @throws Throwable
     */
    public Object traceConnectors(MethodInvocation invocation) throws Throwable {

        Request req = null;
        try {
            Method method = invocation.getMethod();
            CommonInteraction ci = (CommonInteraction)invocation.getTargetObject();

            // For the type we keep the class name without the package path...
            String[] splittedClassName = ci.getClass().getName().split("\\.");
            String requestType = splittedClassName[splittedClassName.length-1];

            req = createRequest(invocation, ReferenceNames.CONNECTOR, requestType);
            CommonManagedConnectionFactory cf =
(CommonManagedConnectionFactory)ci.getManagedConnection().getManagedConnectionFactory();
            req.setValue(ReferenceNames.REMOTE_HOST, cf.getServerName());
            req.setValue(ReferenceNames.REMOTE_PORT, cf.getPortNumber());
            reqDao.saveAllValues(req);
            return invocation.invokeNext();
        } catch (Exception e) {
            reqDao.setFailed(req, checkMsgNotEmpty(e));
            throw e;
        } finally {
            reqDao.close(req);
        }
    }



    /**
     * Advice on the Workflow entry point pointcut.
     * @param invocation
     * @return
     * @throws Throwable
     */
    public Object traceWorkflow(MethodInvocation invocation) throws Throwable {
        Request req = null;

        Object[] args = invocation.getArguments();
        if ((args!=null) && (args[0]!=null)) {
            for (int i=0; i<args.length; i++) {
                if ((args[i]!= null) && (args[i].getClass().toString().indexOf("Long")>=0)) {
                    Long definitionId = (Long)args[i];
                    ExecutionServiceImpl execServImpl =
(ExecutionServiceImpl)invocation.getTargetObject();
                    Definition definition = execServImpl.getDefinition(definitionId);

                    req = createRequest(invocation, ReferenceNames.WORKFLOW,
definition.getName());
                    break;
                }
                req = createRequest(invocation, ReferenceNames.WORKFLOW,
ReferenceNames.UNDEFINED);
            }
        } else {
            req = createRequest(invocation, ReferenceNames.WORKFLOW,
ReferenceNames.UNDEFINED);
        }
        try {
            return invocation.invokeNext();
```

```
        } catch (Exception e) {
            reqDao.setFailed(req, checkMsgNotEmpty(e));
            throw e;
        } finally {
            reqDao.close(req);
        }
    }


    /**
     * Advice on the Workflow Action entry point pointcut.
     * @param invocation
     * @return
     * @throws Throwable
     */
    public Object traceFlowAction(MethodInvocation invocation) throws Throwable {
        Request req = null;
        try {
            Object[] args= invocation.getArguments();
            if ((args!=null) && (args[0]!=null)) {
                ExecutionContextImpl eci = (ExecutionContextImpl)args[0];
                req = createRequest(invocation, ReferenceNames.WORKFLOW_NODE,
eci.getDefinition().getName());

req.setValue(ReferenceNames.FLOW_ACTION_DESC,eci.getDefinition().getDescription());
            } else {
                req = createRequest(invocation, ReferenceNames.WORKFLOW_NODE,
ReferenceNames.UNDEFINED);
            }

            reqDao.saveAllValues(req);
            return invocation.invokeNext();
        } catch (Exception e) {
            reqDao.setFailed(req, checkMsgNotEmpty(e));
            throw e;
        } finally {
            reqDao.close(req);
        }
    }


    private Request createRequest(MethodInvocation methodInvocation, String component, String
type) {
        return reqDao.create(Thread.currentThread().toString(),
                component,
                type,
                methodInvocation.getTargetObject().getClass().getName(),
                methodInvocation.getMethod().getName());
    }


    private Request createRequest(ConstructorInvocation constructorInvocation, String
component, String type) {
        return reqDao.create(Thread.currentThread().toString(),
                component,
                type,
                constructorInvocation.getTargetObject().getClass().getName(),
                "constructor");
    }


    /**
     * Return a predefined Error message if the Exception.getMessage() gives null.
     * @param e The Exception to check.
     * @return The error message to store.
     */
    private String checkMsgNotEmpty(Exception e) {
        String msg = e.getMessage();
        if (msg==null || msg.length()==0) {
            return ("No message set in "+e.getClass());
        }
        return msg;
    }
}
```

# Bibliographie


[AspectJ 05] Site web: aspectj project, http://eclipse.org/aspectj/ (2005) (Dernier accès le 05/06/2005).

[AspectJ 05 a] AspectJ Quick Reference http://www.eclipse.org/aspectj/doc/released/quick.pdf (2004) (Dernier accès le 12/07/2005).

[AspectJ 05 b] Aspectj 5: language changes, http://www.eclipse.org/aspectj/doc/next/adk15notebook/ltw.html#ltw-introduction (2004) (Dernier accès le 12/07/2005).

[AOSD 05] Site web: Aspect Oriented Programming Development conference, http://www.aosd.net (2001) (Dernier accès le 18/05/2005).

[Bodkin 05] Bodkin R., Load Time Weaving with AspectJ 5 Load Time Weaving with AspectJ 5, http://rbodkin.blogs.com/ron_bodkins_blog/2005/05/load_time_weavi.html (2005) (Dernier accès le 05/06/2005).

[CGLIB 04] Site web: Code Generation Library, http://cglib.sourceforge.net/ (2004) (Dernier accès le 18/07/2005).

[AspectWerkz Bench 04] AOP Benchmark, http://docs.codehaus.org/display/AW/AOP+Benchmark (2004) (Dernier accès le 20/08/2005).

[Czarnecki 97] Czarnecki K., Eisenecker U. W., Steyaert P., Beyond Objects: Generative Programming, http://www.prakinf.tu-ilmenau.de/~czarn/aop97.html (1997) (Dernier accès le 20/11/2004).

[Eckel 00] Eckel B., *Thinking in Java*, Prentice Hall, New Jersey, 2000.

[eXo platform 04] Site web: eXo platform™, http://www.exoplatform.com (2004) (Dernier accès le 21/08/2005).

[Gamma et al. 95] Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, Boston, United States, 1995.

[Hannemann et al. 02] Hannemann J., Kiczales G., Design Pattern implementation in Java and AspectJ, http://www.cs.ubc.ca/~jan/papers/oopsla/oopsla02-patterns.pdf (2002) (Dernier accès le 20/08/2005).

[Heymans 04] Heymans P., Analyse et Modélisation de Systèmes d'Information, syllabus de cours pour les Facultés Universitaires Notre-Dame de la paix, Namur, 2004-2005.

[Hibernate 05] Site web: Hibernate, http://www.hibernate.org.

[JBoss 01] Burke B., Fleury M., JBoss-AOP forum, ClassProxy's for everything, http://www.jboss.org/index.html?module=bb&op=viewtopic&t=31212 (2001), (Dernier accès le 20/07/2005).

[JBoss 05] Site web: JBoss Developer Zone, http://www.jboss.org (2005) (Dernier accès le 05/06/2005).

[JBPM 05] Site web: JBPM, http://jbpm.org/ (2005) (Dernier accès le 21/08/2005).

[Kersten 02] Kersten M., AO Tools: State of the (AspectJ™) Art and Open Problems, http://www.cs.ubc.ca/~murphy/OOPSLA02-Tools-for-AOSD/position-papers/kersten.ppt (2002) (Dernier accès le 22/11/2004).

[log4j 05] Site web: LOG4J Logging Services http://logging.apache.org/log4j/docs/ (2005) (Dernier accès le 06/07/2005).

[Kiczales et al. 97] Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier J.-M., Irwin J., Aspect-Oriented Programming, http://www.cs.ubc.ca/~gregor/papers/kiczales-ECOOP1997-AOP.pdf (1997) (Dernier accès le 19/11/2004).

[Oracle 05] Site web: Oracle, http://www.oracle.com (2005) (Dernier accès le 21/08/2005).

[Pawlak et al. 04] Pawlak R., Retaillé J.P., Seinturier L., *Programmation orientée aspect pour Java/J2EE*, Eyrolles, Paris, France, 2004.

[Spring 05] Site web: Spring Framework, http://www.springframework.org/ (2005) (Dernier accès le 18/07/2005).

[Spring 05 a] Spring AOP capabilities and goals, Spring - Java/J2EE Application Framework - Reference Documentation, http://static.springframework.org/spring/docs/1.1.5/reference/aop.html#aop-introduction-spring-defn (2005) (Dernier accès le 18/07/2005).

[Struts 05] Site web: Struts, http://struts.apache.org/ (2005) (Dernier accès le 21/08/2005).

[Suzuki et al. 99] Suzuki J., Yamamoto Y., Extending UML with Aspects: Aspect Support in the Design Phase, http://www.cs.umb.edu/~jxs/pub/ecoop99.pdf (1999) (Dernier accès le 06/07/2005).